PORTLISP : A LISP SYSTEM WITH A TRANSLATOR FROM LISP TO C

A Thesis Submitted
In Partial Fulfilment of the Requirements
for the Degree of

MASTER OF TECHNOLOGY

by

ARUNABHA GHOSE

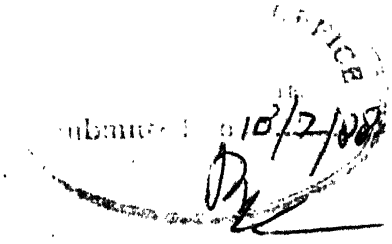
to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

FEBRUARY, 1988

CSE-1988-M-GHO-POR



CERTIFICATE

This is to certify that the thesis entitled 'PORTLISP: A LISP SYSTEM WITH A TRANSLATOR FROM LISP TO C' is a report of the work carried out under my supervision by Arunabha Ghose, and that it has not been submitted elsewhere for a degree.

Karnick.

KANPUR
February, 1988.

Dr. H. Karnick,
Assistant Professor,
Department of Computer Science and Engineering,
Indian Institute of Technology
Kanpur.

ACKNOWLEDGEMENT

It gives me great pleasure to put on record my feeling of gratitude to Dr. H. Karnick, my thesis supervisor. He suggested the problem and created an atmosphere of freedom in which it was a pleasure to work.

I am indebted to Dr. R. Sangal and Debasish Chakroborty who helped me to get a number of useful references.

Special words of gratitude are due to my classmates (M.Tech, CS 1986-87) and residents of Hall 4 without whom I.I.T-K would have been 18 months of boredom.

ABSTRACT

This thesis describes the implementation of a Lisp system with a translator from Lisp to C. The translator is a substitute for a Lisp compiler which directly produces machine code of the target machine. The translator translates source files of Lisp functions and produces optimized C code analyzing the expressions properly. This code can be further compiled using a C compiler and then linked to the Interpreter after which they can be called from interpreted functions within the Interpreter. The system uses its own stack for making frames for calling functions, passing arguments and returning values. Tail-recursion is implemented in the Interpreter and in the compiled code in an efficient way to save time and to avoid stack overflow. The basic modules in the system are - an Evaluator (the core of the interpreter), a Translator, a Reader and a Memory Management module (including a Garbage Collector). The advantage of using this system over others is that the whole system and software developed on it can be ported to any system that supports C. The system implements the 'FRANZ LISP' dialect.

CONTENTS

CHAPTER 1

INTRODUCTION

Lisp is widely used for programming in artificial intelligence appplications. The language is attractive due to its simple syntax, powerful list processing ability, functional behaviour and a good development environment. Lisp uses its symbolic list structure to represent source programs internally and this has no parallel in other languages. Lisp implementations provide interactive interpreters with powerful debugging tools which help users to develop programs efficiently.

There are many dialects of Lisp developed at different places. The users of the Computer Science Departmant of I.I.T., Kanpur were using mainly CLISP avialable on DEC 1Ø9Ø. Lisp was not yet available on the departmental mini computers. It was felt that a Lisp system should be developed which is portable and can be provided on different machines. This thesis describes the implementation of a Lisp system with an interpreter and a translator from Lisp to C. The translator is an experiment in language design and implementation. Unlike the conventional compilers producing assembly language of the target machine directly, the translator produces C intermediate code.

The code for the whole system (PORTLISP) is written in C. As C is available on most of the machines nowadays, PORTLISP can be ported to any such machine. Moreover, Lisp programs written on one machine can be easily transferred to a new machine,

ranslated to C code using the translator and then compiled using he C compiler available on the target machine. As C code gets ompiled efficiently, the efficiency of the final code produced y the C compiler is comparable to that produced by a Lisp com- iler.

.1. FEATURES OF THE IMPLEMENTED LANGUAGE

An interpreter to execute source code without compilation.

Built-in dynamic storage management with a garbage collector to handle the memory allocation efficiently.

Dynamic scoping.

Typeless variables. Any Lisp variable may have as its value any Lisp object.

Powerful macro facility by which the language can be extended and new notations and abstractions developed.

Any mixture of compiled and interpreted code can be run.

Dynamic non-local exits for user-controllable error handling.

Tail-recursive feature. Recursive procedures of a certain form have iterative behaviour.

.2. OVERVIEW OF THE THESIS

In chapter 2, the different data types available in PORTLISP ave been discussed. In chapter 3, a brief description of the isp syntax has been given and the data structures to implement he language have been discussed. In chapter 4, the implementa-

tion of the evaluator, its evaluation procedure and function calling mechanism have been discussed. In chapter 5, a description of the translator and the code segments produced for different Lisp constructs and functions is given. In chapter 6, the functioning of the table driven Lisp reader is discussed. In chapter 7, different I/O functions have been discussed. In chapter 8, the memory management scheme and the garbage collection procedure have been described. Chapter 9 summarises the work done and briefly discusses possible improvements.

CHAPTER 2

DATA TYPES

The implemented Lisp system (PORTLISP) provides a vareity of data objects. In Lisp it is data objects that are typed, not variables. Any variable can have any Lisp object as its value. The following are the different data types.

2.1. LISPOBJ

This is in fact a pointer to any other LISP object. In LISP whenever arguments are passed to a function, a function returns a value or a variable is assigned a value, pointers to those objects not the objects themselves are passed, returned or copied. So this is the most widely used data object, though the user can not directly operate on them.

2.2. SYMBOL

Symbols are named objects which corresponds to variables in other programming languages. Symbols are stored in a hashed array of buckets. Internally, a symbol is represented as a structure with the following fields:-

print_name:

This field is used for storing a pointer to a sequence of characters representing the print name of the symbol. This is used to locate a symbol in the hashed array of buckets and to print a symbol.

value_cell:

This may contain any valid lisp object if the symbol is bound. If the symbol is not bound it contains a special flag indicating that the symbol is unbound.

propety_list:

This field contains a list of an even number of elements, considered to be grouped as pairs. The first element of the pair is the name of the property and the second is the value of that property.

function:

This field contains a function definition, if there is any. It is set by 'def' and is used whenever the function defined by the symbol is applied.

discipline:

This field stores the type of the functional object stored in the function field.

next:

A pointer to the next element in the same bucket of the hashed array.

2.3. LIST

Lists are represented in the form of linked cells called conses. A list cell has two parts called car and cdr. A special object called 'nil' is often treated as an empty list.

2.4. FIXNUM

Fixnums are integer constants in the range -2[31] to 2[31] -

1. One fixnum occupies 4 bytes.

2.5. FLONUM

Flonums are double precision real numbers and occupy 8 bytes each.

2.6. STRING

A string is a null-terminated sequence of characters. A sequence of characters surrounded by double-quotes is recognised by the reader as a string.

2.7. PORT

Ports are used for all stream I/O operations. A port is a structure consisting of two fields.

name:

A pointer to a string denoting the name of the file opened by the functions fileopen, infile, or outfile.

fp:

A file-pointer returned by the 'c' routine fopen(). As a process in the UNIX system can have only twenty files open at a time, there are only twenty ports which are allocated by fileopen, infile or outfile and deallocated by close.

2.8. FCLOSURE

The function fclosure creates this data object which is a valid functional object. Internally, it is a structure with the following fields.

clos_func:

Stores the functional object of the closure.

bindno:

Number of bindings saved inside the fclosure.

clos_bind:

A pointer to an array of structures. Each structure has two fields: one is used to store a pointer to the value-cell of a symbol and another is used to store the value of that symbol when the fclosure was created. The number of elements in the array is exactly equal to bindno.

CHAPTER 3

OVERVIEW OF THE LISP SYNTAX AND THE STACK MACHINE

Lisp programs are organised as 'forms' or 's-expressions and functions. S-expressions are evaluated to produce values and side-effects. Functions are invoked by applying them to arguments.

3.1. S-EXPRESSIONS

The standard unit of interaction with a Lisp system is the s-expression, which is simply a data object meant to be evaluated as a program. Expressions can be divided into three categories : self-evaluating expressions, symbols and lists.

3.1.1 SELF-EVALUATING EXPRESSIONS

All fixnums, flonums and strings are self-evaluating expressions. When such an object is evaluated, the object itself is returned as the value of the expression. The empty list '()', which is the false value 'nil' and the true value 't' are also self-evaluating expressions.

3.1.2 VARIABLES: ITS SCOPE & BINDING

Symbols are used as names of variables. Variables can be assigned to, as by 'setq', or bound, as by 'let' or a function-call. PORTLISP uses dynamic scoping of variables i.e. when a symbol, free in a function body is evaluated, it produces the value of that variable in the environment of the calling func-

tion. There are two strategies for implementing dynamic scoping - shallow binding scheme and deep binding scheme. In deep binding, the stack is searched downwards through a chain of frame pointers to find the most recent binding which is available only in the frame of a function, which was called earlier and created a binding for that variable. In shallow binding scheme, every symbol has a value-cell where the most recent value is available. Whenever a new binding for a variable is created, a pair consisting of a pointer to the value-cell and the content of the value-cell is saved on a stack called 'old binding stack'. When the current environment is exited, the old value is restored to the value-cell. So, whenever a symbol is evaluated the content of the value-cell is produced. PORTLISP uses the shallow binding scheme. While the lookup time in deep binding can be arbitrarily large, if the variable is located deep below the stack, in shallow binding it is small and constant.

However, in deep binding the context switching is fast while in shallow binding the context switching is slow due to save and restore operations. Actually, only a few variables are used free in functions, and few of the bound variables are intended to be accessed by other functions. So in the compiled code, only the variables declared as 'specials' are shallow bound. Others are put only on the stack to avoid the save and restore operations. These are accessible only to the function which created them.

3.1.3. LISTS

When a list is evaluated the first step is to examine the first element of the list. If that is a functional object or a symbol which evaluates to a functional object, the functional object is applied to the other elements of the list after evaluating or not evaluating them depending upon the type of the functional object. Different types of functional objects can occupy the 'function' field of a symbol. Table 3.1 shows the different possibilities.

name of functional object	object type
interpreted lambda function	list with car eq to lambda
interpreted nlambda function	list with car eq to nlambda
interpreted lexpr function	list with car eq to lexpr
interpreted macro	list with car eq to macro
compiled lambda function	pointer to compiled code
compiled nlambda function	pointer to compiled code
compiled lexpr function	pointer to compiled code
compiled macro	pointer to compiled code

Table 3.1

3.2.FUNCTIONAL OBJECTS

3.2.1. FUNCTIONS

There are three types of functions implemented in PORTLISP. The basic Lisp function is the lambda function. When a lambda function is called, the actual arguments are evaluated from left to right and are lambda bound to the formal paramaters of the lambda function.

When an nlambda function is called, the list of unevaluated arguments is lambda bound to the single formal parameter of the nlambda function. An nlambda function is used to write functions which want to evaluate their arguments in special ways.

Lexpr functions are used to handle variable number of arguments. The number of actual arguments passed to a lexpr is bound to the single formal parameter of the lexpr. The arguments can be accessed or set by the called function using 'arg' or 'setarg' respectively.

3.2.2. MACROS

If an expression is a list and the first element is a macro expression or the name of a macro, it is said to be a macro call. Macros are essentially like functions but at least two evaluations take place in case of a macro call. In the first phase, the whole list is bound to the single parameter of the macro and the macro is called. The macro returns a new Lisp expression which takes the place of the old expression. This action is called macro expansion. In the second phase, this new expression is again evaluated to get the ultimate value. In the first phase the expanded expression can again be a macro call. In that case repeated macro expansions will take place.

3.2.3. FCLOSURES

Fclosures are a type of functional object. The purpose of using fclosures is to save the values of some variables between invocations of a functional object and to protect these values from being inadvertently overwritten by other functions. Fclosures are created by the function also called 'fclosure'.

When an fclosure is funcalled,

1) The Lisp system lambda binds the symbols in the fclosure to their values in the fclosure.

2) It continues the funcall on the functional object of the fclosure which itself can be another fclosure.

3) After the funcall is over, the symbols in the fclosure are lambda unbound and at the same time current values of the symbols are stored back into the fclosure. If more than one fclosure was funcalled, this operation lambda unbinding takes place in the reverse order.

3.3. STACKS

PORTLISP uses its own stacks to implement the function calling mechanism for compiled and interpreted functions and to implement dynamic scoping. There are three stacks for this purpose : 'Control Stack' (CS), 'Binding Stack' (BS), and 'Code Stack' (CDS). Control stack is used to keep track of flow of control, whereas the old bindings of symbols are stored on the binding stack. Code stack is used only by the evaluator, which places the code which is being or to be executed on it.

3.3.1. CONTROL STACK

All the activation frames for function calling are made on the control stack. There are two types of frames - 'activation frames' holding informations about function calls and 'catch frames' holding informations about non-local dynamic exits. All the actual parameters for a called function are passed on the control stack and the value from the called function is also returned on the control stack. Besides, the temporary variables for a function are allocated on the control stack - just above the activation frame of the function. The control stack is accessed by two pointers 'Control Stack Pointer' (CSP), pointing to the first free element on the stack and by different types of frame pointers, explained later.

3.3.2 BINDING STACK

Whenever a symbol is lambda bound, the value in its value-cell is stored on the binding stack. When the current environment is left, the old value on the stack is restored in the value-cell. 'BSP' points to the first free element on the binding stack. Each element on this stack is a structure of two elements:

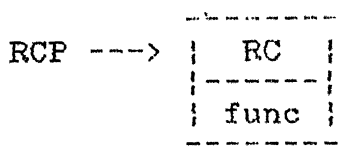
adr: A pointer to a value-cell.

bind: The old value of that value-cell.

3.3.3 CODE STACK

This stack is used only by the evaluator. It is used to store the codes which are being interpreted. For each frame the

evaluator reserves two elements on this stack as shown below. 'RCP' points to the topmost element on this stack.



Non-active frame (only when the function is being interpreted)

RC: A list containing the rest (unevaluated arguments).

func: A functional object, which is to be applied.

Active frame

RC: A list containing the rest of the body of the function being applied.

func: A functional object, which is being applied.

3.3.4. FRAME STRUCTURE

There are two types of frames used by PORTLISP - 'activation Frame' and 'Catch Frame'. There is one more type of frame, called 'Non-active Frame' used only by the evaluator. A non-active frame is created when the evaluator starts evaluating any list. After evaluating the arguments (in some cases they may not be evaluated, as for nlambda and macro), this frame is given 'active' status. 'NAFP' is a pointer to the most recent non-active frame and 'AFP' is a pointer to the most recent active frame. After creating the activation frame the function is called. A catch frame is created only when the function 'catch' is executed. 'CFP' points to the most recent catch frame.

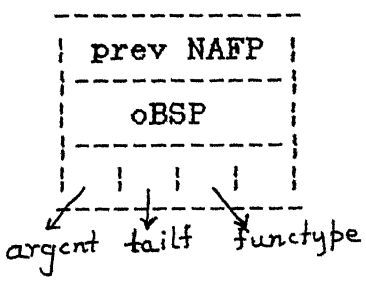


Fig 3.2(a) Non-activated frame

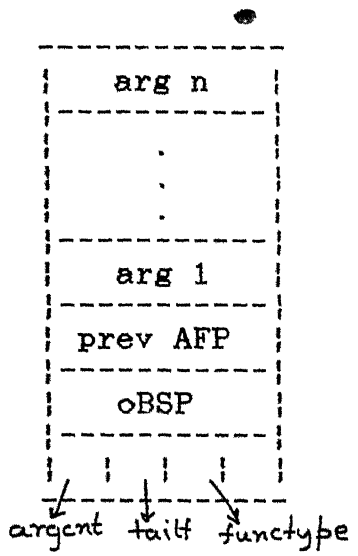


Fig 3.2(b) Activated frame

ACTIVATION FRAME FORMATS

argcnt: Number of arguments passed to the called function.

tailf: A flag that indicates whether the function call is tail recursive or not. It can have following values:

Ø - The call is not tail recursive.

1 - A function is calling itself in a tail recursive manner.

2 - A function is calling another function in a tail recursive manner.

functype: An integer which denotes the type of function call The following types are possible.

1) LAMBDA - Interpreted lambda call.

2) NLAMBDA - Interpreted nlambda call.

3) LEXPR - Interpreted lexpr call.

4) MACRO - Interpreted macro call.

5) CLAMBDA - Compiled lambda call.

6) CNLAMBDA - Compiled nlambda call.

7) CLEXPR - Compiled lexpr call.

8) CMACRO - Compiled macro call.

9) NR_CLAMBDA - Compiled built-in function for which the frame manipulation is never tail recursive even if the call actually is.

oBSP: A pointer to the binding stack. It points to first free element on the binding stack when the frame is activated.

3.4. RETURNING VALUES

All the lisp functions return values even if only the side effects are of interest. In PORTLISP functions return values through the control stack. When a function returns, it puts the return value at the lowermost position of its activation frame and sets the control stack pointer to the element just above it. All the elements starting from this position are free after the function call is over. The calling function retrieves the return value using the control stack pointer. This strategy has been used because most of the times functions are called to set up the argument list for another function call and so the values of expressions are expected on the top of the stack. So values are automatically put at the places where they are expected and thus copying from one location to another is avoided.

CHAPTER 4

THE EVALUATOR

The evaluator is at the centre of all the activities of a Lisp system. All interpretations of expressions are done by the evaluator which takes expressions, checks them, decides the actions to be taken and executes them. The recursive definition of an evaluator is given below.

```
function eval(LISPOBJ exp): LISPOBJ;
begin
   if exp is an atom then
     if exp is self-evaluatable /* fixnum/flonum/string */
       return exp;
     else                       /* exp is a symbol */
       if exp is bound          /* some value is present in the
                                    value cell of the symbol */
         return value stored in the value cell;
       else
         return error;          /* unbound symbol */
   else                         /* exp is a list */
     if car(exp) is a lambda
       begin
         evaluate each element of cdr(exp) and
         put them in argument-list;
         apply car(exp) on argument-list;
       end;
     else if car(exp) is an nlambda
       begin
         apply car(exp) on cdr(exp);
       end;
     else if car(exp) is a lexpr
       begin
         evaluate each element of cdr(exp) and
         put them in argument-list;
         apply car(exp) on argument-list;
       end;
     else if car(exp) is a macro
       begin
         apply car(exp) on exp;
         evaluate the return value;
         return the result of evaluation;
       end;
     else
```

```
        error;          /* car(exp) is not a functional object */
end;
```

4.1. THE FUNCTION EVAL

However, the actual evaluator used by PORTLISP is an iterative one as that is more efficient. It does not call itself recursively. Calls to the evaluator may be made from user functions calling eval explicitly, or by some in-built functions which evaluate their arguments in special ways.

The evaluator consists of two main modules - 'eavl-expression' (eval-exp in short), and 'eval-frame'. While entering, the function eval remembers the value of the control stack pointer at that point as 'initialCSP'. Control is transferred between the modules as necessary. Inside the modules some computations are done, functions are invoked and stacks are modified. Finally, when CSP comes down to initialCSP, the evaluator exits.

The function eval can be called in two modes. In the first mode, eval is passed an expression to be evaluated. In this case, the expression is evaluated and the result is returned. In the second mode, the calling function makes an activation frame for calling a functional object, puts the arguments at proper places after evaluating them and calls eval passing the functional object. Eval applies the functional object, puts the return value on the top of stack and returns.

MODULE - 1 : eval-expression

This module takes an expression from a variable 'exp'. If exp is an atom, it checks whether it is self-evaluating. If it

is, exp is put on top of the control stack. If it is a symbol, the content of the value-cell of the symbol is put on top of the stack. CSP is incremented and control goes to eval-frame. If exp is a list, a non-active frame is created on the control stack. The functional object is checked and accordingly a list of unevaluated arguments is put on the code-stack. In case of nlambda and macro 'nil' is put on the code-stack as arguments of these two are not evaluated. Next, the control is transferred to eval-frame for evaluation of arguments and application of the function. If tail recursion flag is on, the function call is tail-recursive. It is checked, whether tail recursion can be allowed. If the function to be applied is a macro, tail recursion is not allowed as the return value of the macro has to be evaluated in the environment of the calling function. A tail recursive call would remove the bindings created by the calling function and the evaluation would not be proper.

MODULE - 2 : eval-frame

This module is entered to compute the state of the topmost frame on the control stack and to decide the next action. This module initiates evaluation of the arguments for a function call, calls a function after that and decides when to return.

The state of the topmost frame is computed by checking CSP and the three frame pointers AFP, NAFP and CFP and the content of the topmost code in the code-stack pointed by RCP. There are five possible states. The actions taken in those states are described below.

1) CSP is equal to (initialCSP + 1)

The control stack pointer has come down to the value whe eval was entered. The evaluation process is complete and th result is on top of the stack pointed by 'initialCSP'. So, eva must return now. If eval was called in mode one CSP is furthe decremented to initialCSP and the value pointed by it i returned. Otherwise, eval just returns. No value is returned a it is expected by the calling function on top of stack pointed b initialCSP.

2) Top frame is non-active and the top element of the code stac (content(RCP)) is not nil.

In this case, more arguments are to be evaluated before th top frame can be given active status. Exp is set to the first o the remaining arguments (content(RCP)), content(RCP) is set t the cdr(content(RCP)) and eval-exp is entered.

3) Top frame is non-active and the top element of the code stac is nil.

The top frame is ready for a function call as all the argu ments have been evaluated (as content(RCP) is nil). The frame i made active by restoring the old value of NAFP from FP field an storing the current AFP in the same field. AFP is set to poin to the FP field. If this is not a tail recursive call, th present value of BSP is stored in the oBSP field of the frame When a function returns, this is used for restoring old bindings

After giving the frame active status, the function-call

initiated. If the functional object is an fclosure, all the old values stored in the fclosure are put in the value-cells of the corresponding symbols after saving the current values. The current values are saved in the places earlier occupied by the values in the fclosure object.

For a compiled function, the function is called through a pointer to that function stored in the function field of the corresponding symbol entry and the function application is over at this point only. The called function resets AFP and CSP itself. After return of the called function, the return-value is available on the stack. If it was a macro-call, exp is set to the return-value and eval-exp is entered for the evaluation of the macro expanded object. Otherwise, eval-frame is entered again to decide the next action.

In case of an interpreted function, this point is only the beginning of the function application. The top of the code stack [till now it was being used to hold the unevaluated arguments, if there were any] is set to hold the function body and eval-frame is entered.

4) Top frame is active and the top element of the code stack is not nil.

The evaluation of the body of the function currently being applied, is not yet complete. The first element of the rest of the code is taken from the top element of the code-stack (content(RCP)) and it is put in exp. Content(RCP) is set to cdr(content(RCP)). If it is nil and the current function is

either a lambda or an nlambda, the tail-recursion flag is set.

5) Top frame is active and the top element of the code-stack is nil.

This condition holds whenever a function-application of an interpreted function has been finished by the evaluator. The bindings created by the current funtion are removed and the old bindings saved in the binding stack are restored. In the process, the binding stack pointer is decremented to the value stored in oBSP field of the frame. If the applied function is a lambda / an nlambda / a lexpr, the return value is taken from the top of the stack and is put in place of the lowermost element of the frame. The control stack pointer is made to point to the element just above it. If it was a macro-application, the return-value is put in exp for further evaluation. If the applied functional object was an fclosure, the values stored in the fclosure object are interchanged with the values stored in the value-cells of the corresponding symbols. Thus the current values of the symbols in the fclosure are protected till the application of the fclosure next time. In case of macro-application, the control is transferred to eval-exp; otherwise, it is transferred to eval-frame.

4.2. HANDLING TAIL RECURSION BY THE EVALUATOR

When the last expression in the body of a function produces a recursive call to the same function, the process is iterative in nature. The value returned by the called function is also returned from the calling function. As the process is iterative,

the number of elements on the stacks should not grow. When a funtion calls itself in a tail recursive manner, the evaluator uses the frame of the calling function to produce the next call. As for an interpreted funtion, no local variables are present on the stack, the arguments for the next function call are directly put at proper places of the frame overwriting the arguments of the calling function. So the control stack does not grow for a tail recursive call.

After the arguments are evaluated, next function call is initiated. At this time the current value of the binding stack pointer is not stored in the frame as it is already stored by the calling function. The value of BSP stored in the frame is simply inherited by the next call. While the parameters are lambda bound to the arguments, old values are not saved on the binding stack as they are already saved by an earlier call to the function. When the termination condition for a series of tail recursive calls is reached, the last function call puts the return value on the stack and uses the value of the BSP stored by the first call to the function, to restore the old bindings. So saving bindings of parameters is done only at the time of the first call and restoring is done only after the last call. Thus, the binding stack does not grow with tail-recursive calling and lot of save and restore operations are avoided.

When the last expression of a function body produces a call to a different function, the call is not truely tail-recursive as the process is not iterative in nature. Bindings have to be saved when the parameters of the called function are lambda bound. But

some optimizations are possible in this case also. We notice, just before producing the call only the oBSP field is useful as it will be used to restore bindings while returning. So the frame of the calling function is used by the called function. The bindings are restored only when the last of a series of functions using the same frame returns. Here though the binding stack grows, the control stack does not grow.

[]

Let us take a small example to explain the functioning of the evaluator. The definition of function f is given below.

```
(def f (lambda (x y z) (append (list x y) z)))
```

When the evaluator is passed an expression '(f a 2 (g 3 4))', the arguments are first evaluated. Here f and g are interpreted functions, and 'list' and 'append' are in-built functions. Let us assume that the function g returns a list of the two arguments passed to it and 'a' has a binding to the fixnum 1. Lambda expressions of f and g are shown as lambda1 and lambda2 respectively. First, control goes to eval-exp and a non-activated frame for calling f is created and the unevaluated arguments are put on the code stack. The contents of the control stack and the code stack are shown in fig. 1.a. The control goes to eval-frame which sees that the top frame is non-active and the arguments are not yet evaluated. So eval-exp is entered for evaluating each of them. The symbol 'a', and and the fixnum 2 are evaluated in eval-exp only and the values are put on the control stack (fig. 1.b). To evaluate '(g 3 4)', a new non-active frame is created and the unevaluated arguments are put on the code stack (fig.

1.c). After evaluating the arguments of g, the top frame is given active status (fig 1.d). g returns the list '(3 4)' on the stack. As no more arguments of f are left to be evaluated (content of the code stack is nil), the frame for f is made active and the body of f is put on the code stack (fig. 1.e). The body has got only one expression (tail-recursive of type 2). Append is a compiled lexpr and no new frame for it is created; the frame for f is used to call append. First the frame is made non-active and the unevaluated arguments are put on the code stack (fig. 1.f). After the arguments are evaluated (fig. 1.g), the compiled function is called in module 3 of eval-frame using the pointer to the compiled code. The contents of the stack after append returns is shown in fig. 1.h. CSP is further reduced to initialCSP and '(1 2 3 4)' is returned from eval.

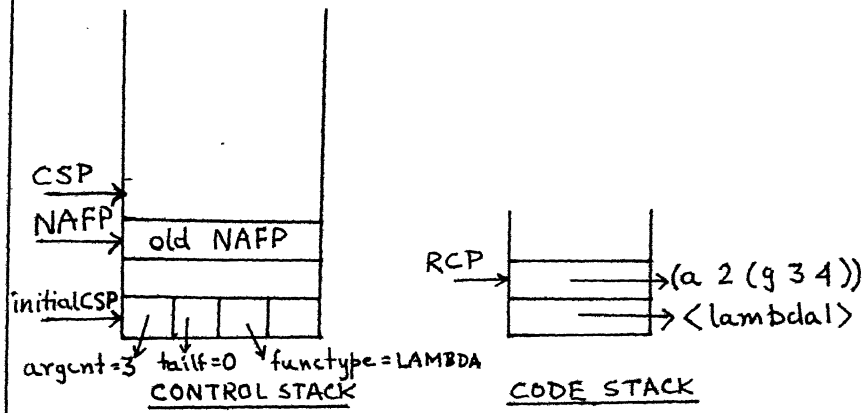


Fig 1.a

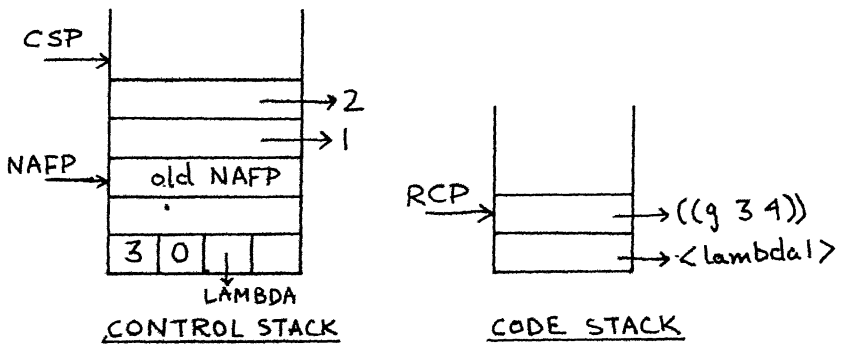


Fig 1.b

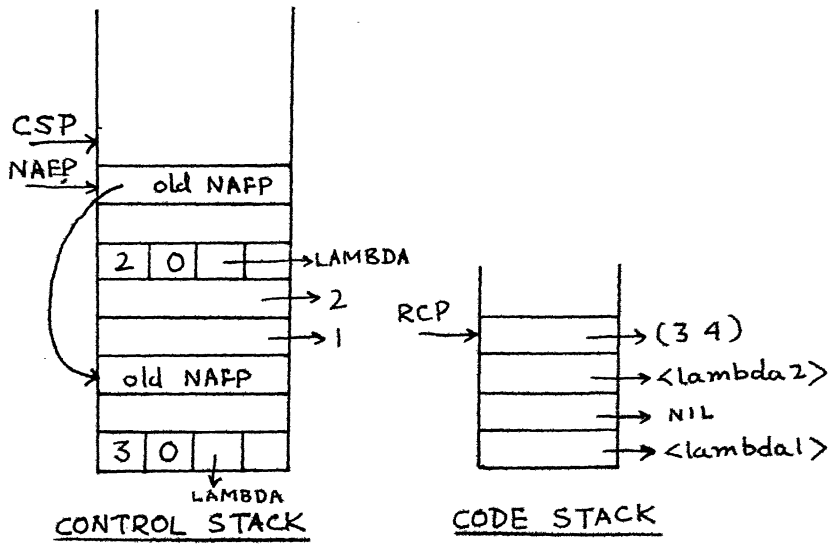


Fig - 1.C

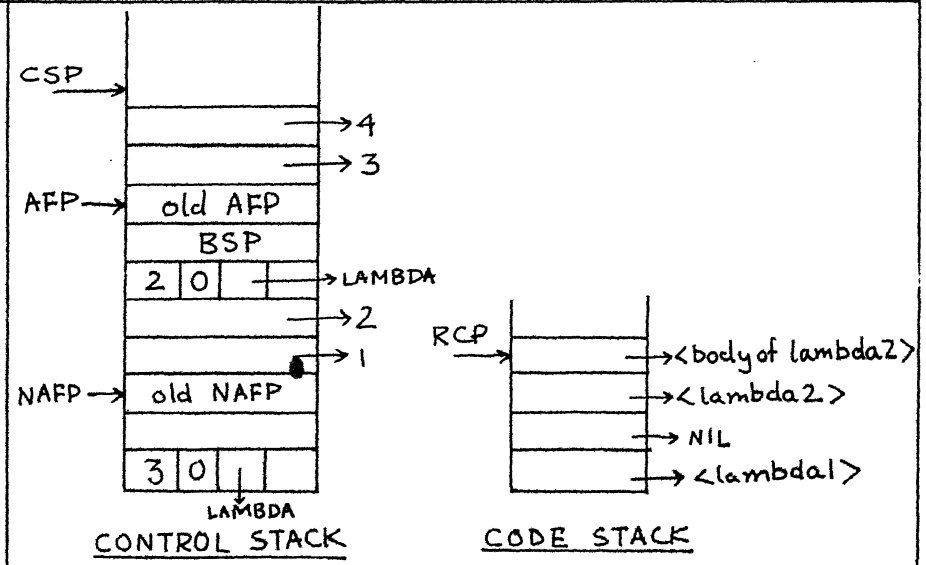


Fig - Ld

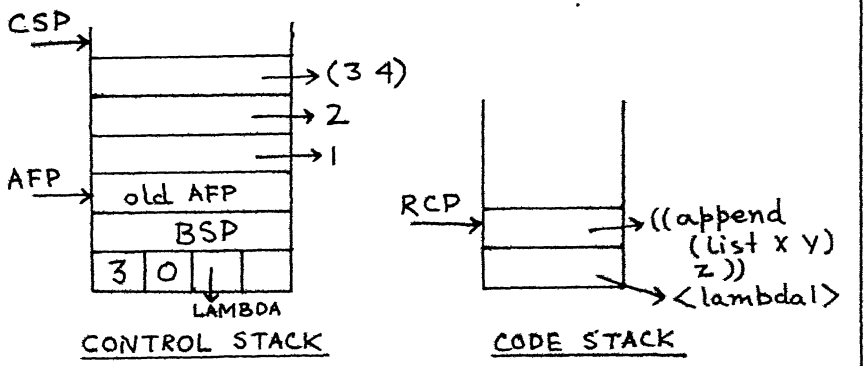


Fig 1.e

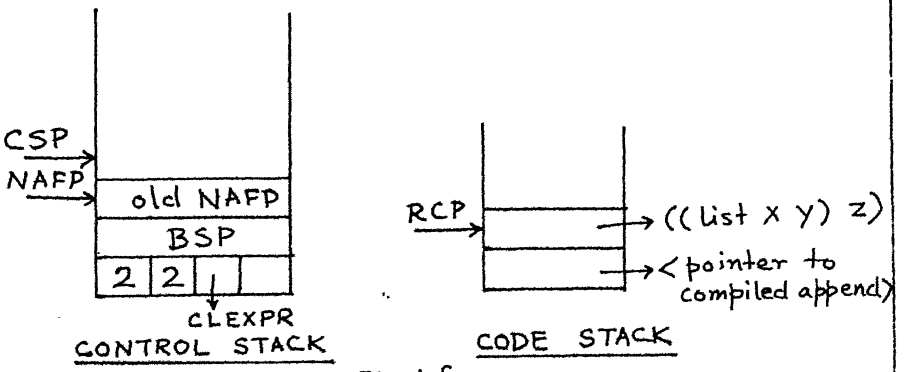


Fig 1.f

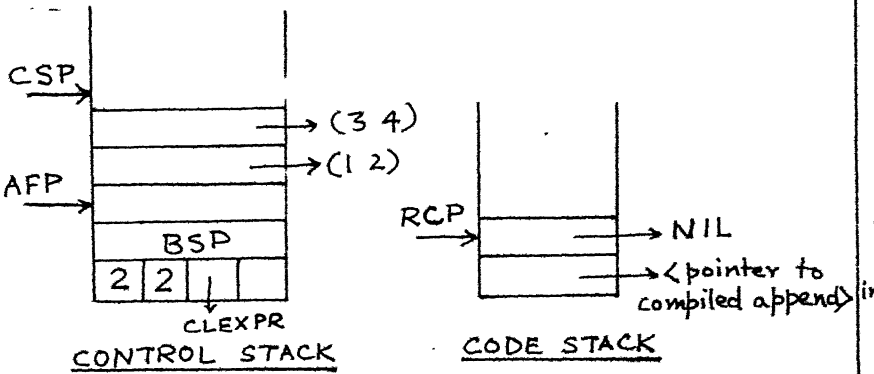


Fig 1.g

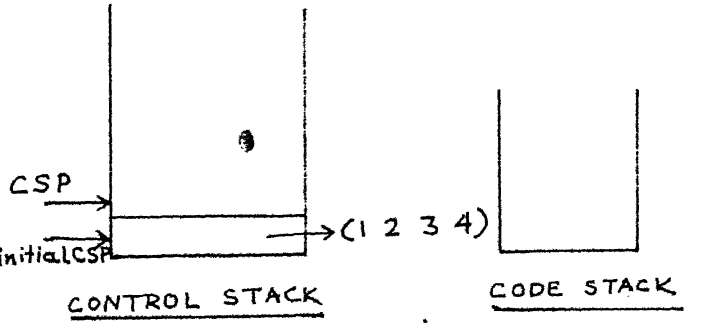


Fig 1.h

4.3. IMPLEMENTATION OF SPECIAL FUNCTIONS

4.3.1. let

It is implemented as a compiled nlambda. Eval passes the cdr of a let expression encountered by it to the in-built let function. At first, let evaluates all the initialization expressions and stores them on the stack. Then it binds those values to the corresponding symbols. This gives the effect of parallel assignment of the values to the symbols. After creating the bindings let puts the body of the let expression on the code-stack and transfers the control back to eval without removing the frame. Eval continues to evalute the body using the same frame which is removed after the evaluation of the let body is over.

4.3.2. let*

It is impemented in almost the same way let is implemented. Here, just after evaluating an initialisation expression it is bound to the corresponding symbol as the assignments to symbols are to be sequential.

4.3.3. prog

It is implemented as a compiled nlambda. At the beginning, every symbol, present in the list of parameters, is lambda bound to nil. As the prog body is evaluated, any symbols seen are not evaluated, instead they are treated as labels. Prog returns the value explicitly given in a return form or nil if no return is done by the time the last expression is evaluated. 'Return' and 'go' can be used only within a prog body and these constructs are

implemented using three globals - a flag called 'returning' and two variables 'returntype' and 'returnval'. These are set whenever a return or a go is executed as discussed in the next two sections. As the prog body is evaluated sequentially from left to right, after each evaluation the returning flag is checked. If it is on, a return or go has been executed. If it is a return [this information can be retrieved from returntype], the value in returnval is put on stack and prog returns removing the frame and restoring the old bindings. If the returning flag has been set by a go, the target of the go (a symbol) is retrieved from returnval and a sequential search is carried out to check whether the label is present in the prog body. If it is present, evaluation starts from the next expression in the prog body. Otherwise, prog returns without disturbing the returning flag, so that the condition can be caught by some other outer prog.

4.3.4. return

It makes the returning flag on, sets returntype to indicate that a return is being done and puts the evaluated argument in returnval to be retrieved by a prog.

4.3.5. go

It makes the returning flag on and sets returntype to indicate that a go is to be executed. The argument of the go (the target of the go) is repeatedly evaluted until a symbol results and this symbol is put in returnval to be retrieved by a prog.

4.3.6. cond

Eval passes a list of predicates (the unevaluated body of a cond expression) to 'cond'. The clauses are processed from left to right. The first element of a clause (the predicate) is evaluted and if it is non-nil, then the clause is said to be satisfied. If no other expression is present in the clause, the value of the predicate is put on the stack as the return-value of the cond and it returns removing the frame. If there are more expressions in the clause, they are put on the code-stack and cond returns without removing the frame. Eval continues to evalute those expressions and the value of the last one is put on the stack as the return-value of the cond. If none of the predicates is satisfied nil is returned.

4.3.7. and

A list of unevaluated arguments are passed to 'and' by the evaluator and it evaluates them sequentially. If all arguments evaluate to non-nil values, the value of the last argument is returned; otherwise nil is returned. It is implemented as an nlambda.

4.3.8. or

A list of unevaluated arguments are passed to 'or' by the evaluator and it evaluates them sequentially. In the process, the value of the first argument, which evaluates to a non-nil value, is returned without evaluating the following expressions; otherwise nil is returned. Like 'and', it is implemented as an nlambda.

4.3.9. def

'Def' puts a new function definition for a symbol. It must be given two arguments (unevaluated) - a symbol and a list which can be applied. The car of the list should be one of lambda, nlambda, lexpr and macro which denotes the type of the functional object. If the functional object is proper, 'def' puts it in the 'func' field of the corresponding symbol.

4.3.1Ø. apply

Apply is passed a functional object and a list of arguments. It puts the arguments in proper places of the frame and calls eval in mode two which does the function application. After the application is over eval puts the return value on the stack and removes the frame. As the control is transferred back to apply, it simply returns.

4.3.11. funcall

Funcall works in the same way as apply works. It shifts the arguments passed excluding the first one [which is a functional object] by one place and calls eval in mode two for applying the functional object.

4.3.12. arg

Arg must be used in the body of a lexpr. It searches the control stack downwards to get the frame pointer to the enclosing lexpr. If arg is not passed any argument, it finds the number of arguments passed to the lexpr from its frame and returns the number. If any fixnum (say i) is passed to 'arg' as an argument,

4.3.9. def

'Def' puts a new function definition for a symbol. It must be given two arguments (unevaluated) - a symbol and a list which can be applied. The car of the list should be one of lambda, nlambda, lexpr and macro which denotes the type of the functional object. If the functional object is proper, 'def' puts it in the 'func' field of the corresponding symbol.

4.3.1Ø. apply

Apply is passed a functional object and a list of arguments. It puts the arguments in proper places of the frame and calls eval in mode two which does the function application. After the application is over eval puts the return value on the stack and removes the frame. As the control is transferred back to apply, it simply returns.

4.3.11. funcall

Funcall works in the same way as apply works. It shifts the arguments passed excluding the first one [which is a functional object] by one place and calls eval in mode two for applying the functional object.

4.3.12. arg

Arg must be used in the body of a lexpr. It searches the control stack downwards to get the frame pointer to the enclosing lexpr. If arg is not passed any argument, it finds the number of arguments passed to the lexpr from its frame and returns the number. If any fixnum (say i) is passed to 'arg' as an argument,

then 'arg' finds the ith argument passed to the lexpr and returns it.

4.3.13. setarg

Setarg is used in the body of a lexpr. It is passed two arguments, the first of which must be a fixnum (say i). It searches the control stack in the same way as arg does to access the frame of the enclosing lexpr, and sets the ith argument of the lexpr to the second argument passed to setarg.

4.3.14. do

Do is implemented as a macro which gets expanded into a prog expression.

4.3.15. *catch

'*catch' and '*throw' have been provided for non-local dynamic exits. With the help of these two functions users can implement their own error-handling routines efficiently.

A *catch is passed two arguments - the first one should be a symbol or a list of symbols (considered as tags), and the second one can be any s-expression. At first, *catch sets up a catch frame on the control stack and stores the current value of BSP, RCP and the first argument in the frame. It executes a call to the system function 'setjmp', and then calls eval to evaluate the second argument. If the evaluation takes place normally, the evaluated value is returned. However, if a value is thrown during the evaluation process, *catch returns with that value if any one of the following cases is true:

1) the tag thrown is equal to the first argument (if it is a symbol).
2) the first argument is a list and the tag thrown is a member of that list.
3) the first argument is nil.

4.3.16. *throw

Throw is passed two arguments - a symbol (tag) and an expression. The value of the second expression is returned from the enclosing *catch with the same tag.

Internally, two system calls - setjmp and longjmp are used to implement *catch and *throw. When a setjmp is done, the current context is stored in a structure called 'jump-buffer'. Later when a longjmp is done on the same jump buffer, the previous context is restored. Here an array of jump buffers has been used to synchronize the operations of *catch and *throw. Whenever a *catch is done, the context is stored in the next element of the array of jump-buffers. If *catch returns normally, it removes the jump-buffer created by it. If a *throw is done (it is always done on the most recent jump-buffer), the innermost *catch checks the tag thrown. If the tag matches, the whole control stack upto the catch frame, the code stack upto the value of RCP stored in the catch frame and the binding stack are cleaned (the C stack gets cleaned automatically by the call to longjmp). If the tag does not match, *catch executes another longjmp on the previous jump-buffer so that the next enclosing *catch can check the condition.

CHAPTER 5

THE TRANSLATOR

The translator translates source programs written in Lisp into equivalent source programs in C. The files containing C functions can be compiled and linked to the basic modules of the Lisp system. The translator (we call it LISPC) is much like an interpreter both in its structure and in the function it performs. The mechanisms used by the translator for analyzing expressions are similar to those used by the interpreter.

The generated code uses the same control stack and binding stack for calling functions. The structure of the frame for calling functions from translated functions is also same. This helps to interface the evaluator and translated code properly.

Our ultimate objective to translate Lisp code to C code and compile the C code is to get object code in machine language which runs faster. The translator carries out a number of optimizations by analyzing the source code. All the constructs like prog, cond etc are implemented in the interpreter using special functions. But LISPC open-codes many such constructs to make the code faster. As more open-coding is introduced, the code becomes more efficient. These will be discussed in later sections.

As LISPC reads a source file, all the 'def' expressions (function definitions) present at top level are translated to corresponding C functions. All other expressions present at top

level are evaluated to build up a translation time environment. As the arguments are passed on the control stack, the generated C functions do not have any formal parameters. The generated code is produced in the form of functions to keep the modularity of Lisp files intact and files can be translated and compiled separatly and then linked. Also, this makes the transfer of control to the code of callee easier as in a high level language like C it is not possible to operate on addresses of instructions [to implement a function calling mechanism it is necessary to push and pop return addresses].

5.1. HANDLING DECLARATIONS

Declare is used to provide information about functions and variables to the translator. As arguments to different types of functions are treated in different ways, the translator must be provided with informations about the types of the functions being called in a file. A function can be declared any one of the three types: lambda, nlambda and lexpr. A called function can be either interpreted or compiled. As the calling mechanisms for interpreted functions and compiled functions are different, this information should also be available to the translator. If nothing is mentioned, the default type for the called function is assumed to be compiled lambda.

If there is any macro call in a file and the macro definition is not present in the file, the translator must be notified about it by using the 'macro' declaration. Normally, the macro definitions in a file are not translated, they are used to expand the macro calls at the time of translation. If the variable

'macros' is set to a non-nil value or if the symbols denoting the names of the macros are declared using 'macros' then the macro definitions are also compiled. This can be done by '(declare (macros t))' or by '(declare (macros [macro-names]))'.

Variables may be declared special or unspecial. Whenever a special variable is lambda bound, its old value is stored on the binding stack and the new value is put in its value-cell. After the duration of the lambda binding, the old value is restored. These save and restore operations take time and often are not necessary for variables which are meant to be local to the function in which they are defined. So, the default classification for variables is unspecial. Space for unspecial variables is dynamically allocated on the stack and they can only be accessed from within the function where they are defined. The user must be careful in declaring the variables. Any variable which is meant to be accessed by other functions through dynamic scoping used in Lisp, must be declared as special; otherwise, its value will not be avialable in the value-cell of the symbol.

Following are the different key-words used for declarations:

lambda	Compiled lambda
nlambda	Compiled nlambda
lexpr	Compiled lexpr
macro	Compiled macro
i-lambda	Interpreted lambda

i-nlambda Interpreted nlambda

i-lexpr Interpreted lexpr

i-macro Interpreted macro

macros Instructs translator to translate the macro definitions present in the declaration.

special Special variables

unspecial Unspecial variables

If no key-words are present in any declarations, they are simply evaluated by the translator. Declarations can be put anywhere in a file at top level (not within any function definitions). So, the same variable can be declared as special in some portions of a file while as unspecial in other portions.

5.2. CREATING BINDING FOR VARIABLES AND ACCESSING VALUES

Depending on the declarations, a variable can be either special or unspecial at a particular time. Whenever the binding for a special variable is created, the old value is stored on the binding stack and the new value is put in the value-cell of the symbol. When the variable is referred, its value-cell is accessed. This value is global and any function can refer to it.

But, if the variable is unspecial, these operations are not necessary. The value is put on the stack and the value can be accessed only within the function which created it. Here the variable name is eliminated from the translated code. As a new binding for an unspecial variable is created, the translator notes down the offset of the variable with respect to the frame

pointer and stores it in the 'offset' field of the symbol entry. Before storing the new offset, it saves the old offset on a special stack called 'offset stack'. When the scope of the current binding is removed, the current value of offset in the symbol entry is removed restoring the old value from the offset stack. During the lifetime of a variable new frames might be created by let, prog or some lambda binding. The variable is still accessible from the prog, let or lambda body through a negative offset from the new frame pointer. So, just before producing code for making a new frame for prog, let etc., the translator updates all the existing offset fields of the unspecial variables by subtracting the offset of the new frame pointer from the old frame pointer. When the translation of the prog or let is over, the same offset of the frame pointers is added to the offset fields to get the old values back.

For example, let us consider the creation of binding for a variable x. If x is special, the code is

```
[1] BSP->adr = &(x.symval->valcell);          /* save the pointer
                                to the value-cell for symbol x */
[2] (BSP++)->bind = x.symval->valcell;        /* save the content
       of value-cell and increment the binding stack pointer. */
[3] x.symval->valcell = *(LAFP + <var_offset>); /* assign the new
           value in value-cell. LAFP is the current stack pointer
           and <var_offset> is the offset of the element from the
     current stack pointer where the value is already present. */
```

fig. c-1

The value can be accessed by 'x.symval->valcell'.

In case of unspecial variables the value is accessed by the expression '*(LAFP + <var_offset>)'.

5.3. HANDLING OF CONSTANTS AND QUOTED EXPRESSIONS

All the self-evaluatable objects i.e. fixnums, flonums and strings are treated as constants. In PORTLISP all the data objects are allocated space at runtime. So, constants in the translated code are handled in a special way. The quoted expressions and the objects which are not to be evaluated [e.g. the argument to an nlambda call], are also treated like constants. For every file, a special function is created to initialize all the constants just after the loading of the object module. This function is named as 'init_cons_<filename>'. It creates all the constants and quoted objects encountered in the file and puts them in an array of Lisp objects - 'ZC[]'. In the translated code these constants are referred through an index into the array 'ZC[]' which is calculated by the translator. For example, if the fixnum 5678 is the third constant encountered while translating a file, it is referred in the translated code as ZC[3].

As the constants are initialized into the ZC array only once after the loading, they are not created every time functions are called. Space is allocated for these constants from static pages, so that these constants are not garbage-collected. Every file has its own ZC array which is local to it and cannot be accessed from functions in other files.

Small fixnums in the range Ø - 1Ø22 are handled in a special way. They are allocated space from a static page while initializing the system and whenever a fixnum in that range is created, a pointer to an object in that page is returned without actually creating the fixnum object. This avoids duplication of small fixnums. The adrress of the fixnum Ø is stored in 'SMALL_FIX_AR'. A

fixnum 123 is referred in the translated code as (SMALL_FIX_AR + 123).

5.4. TRANSLATING FUNCTIONS

A def form is used to define a function. As a file is translated, all the def forms present in the top level are passed to the function 'tran_def' to take proper actions. If the function being defined is a lambda, nlambda or lexpr then 'tran_def' calls 'tran_lambda', 'tran_nlambda' or 'tran_lexpr' to translate the definition into a corresponding C function. If the function being defined is a macro, then the translator evaluates the definition, defining the macro within the translator environment. If the translator has already come across a 'macros' declaration for the macro name, then the macro definition is also translated to a C function to be linked to the object code. C function names are derived from Lisp function names by prefixing them with "F_" as symbol objects will be present by the same names.

When a function definition is translated, macro expansion is done whenever possible. While translating a macro call, if the macro is not already defined within the translator, C statements are produced to expand the macro call at run time.

The body of each function consists of a number of s-expressions. These expressions are compiled recursively calling 'tran_exp'. While translating a function application, if the function is open-coded, tran_exp calls the function which does the open-coding for that particular function. These functions will be discussed later.

When the translation is done, a number of parameters are passed to the translator functions which are called recursively to translate the expressions. The necessity of these parameters are discussed below:

exp: The expression to be translated.

tail: It is a flag which when true, denotes that the expression is tail-recursive i.e. the last statement to be evaluated in the environment produced by the current frame.

offset: It is a constant calculated by the translator which is the offset of the first free element on the control stack with respect to the frame pointer. All the variables on the stack are accessed using the current frame pointer. While translating the expression, all the temporaries generated have offsets more than or equal to this parameter. The elements, whose offsets are less than this value, hold the variables (including temporaries) which are live during the execution of the expression being translated. To make the mechanism of returning values consistent, the generated C statements put the return value of the expression at offset number of positions away from the current runtime frame pointer.

next: It is the label number to which the control should go after the execution of the expression.

cont: It is a flag which if true, denotes that the control should go to the statement present immediately after the

block of statements generated from the current expression. If cont (continue) is false, the last statement produced for the current expression is 'goto <next>'. Otherwise, unnecessary goto statement is not generated.

inlex: A flag which if true, indicates that the accesses on the stack will be made through a pointer through a frame produced for calling a lexpr. The use of this flag will be explained shortly.

In a function body all the variables on the stack are accessed through a frame pointer. Though this value is available in the global variable AFP, the access on the stack becomes slow due one more memory operation. To remove this deficiency each function stores its frame pointer in a local register variable LAFP and this is used for all indirect access on the stack.

The number of arguments passed to a lexpr can not be determined by the translator. Due to this, the offsets of variables will vary at runtime. Just after entering the code for a lexpr, a register variable 'zn' is initialized to the number of arguments passed to it. All the variables accessed through the frame pointer to a lexpr are indexed by the content of zn. If the offset calculated by the translator for a variable is <var_offset>, the variable is accessed by an expression *(LAFP + zn + <var_offset>). All translator functions, while producing code for the access of a variable on the stack through a pointer to a lexpr frame, adds this extra term zn for indirection. In the following sections whenever we show code we will omit this term for convenience.

The format for the code produced for a function is given below.

```
F_<function-name>{
   register *LAFP = AFP;
   [<code to store old values of parameters>] /* for specials */
BEGIN:
   [<code to lambda bind the parameters>]     /* for specials */
   <code to evaluate the function body>
}
```

fig. c-2

5.5. TRANSLATION OF EXPRESSIONS

5.5.1. CONSTANT REFERENCE

An expression can be an atom or a list. If the atom is a constant, i.e. a fixnum, flonum or string, the code produced is

```
[1] *(LAFP + <dest-offset>) = ZC[<index>];
[2] [ goto <next>;]
```

fig. c-3

Here <dest-offset> is the offset of the destination calculated by the translator and <index> is an index into the array of constants for the file. If 'cont' is false, a jump to the label <next> is created. All optional codes will be shown within square brackets.

If the computation of the constant expression is tail-recursive, the code for removing the current frame and bindings are also created. The destination in this case is the lowermost element of the current frame whose offset is -2. In this case, if this is the last expression of a function, the code is

```
[1] *(LAFP - 2) = ZC[<index>];
[2] restorebind(LAFP);                          /* remove bindings */
[3] CSP = LAFP - 1;                  /* set cntrol stack pointer */
[4] AFP = LAFP->ptrval;       /* restore calling frame pointer */
```

```
[5] return;
```

fig. c-4

If the expression is the last in a let, prog or a lambda body being applied, no return should be done. Also, the control stack pointer and the global frame pointer need not be updated. Instead, the local frame pointer is restored. The code is

```
[1] *(LAFP - 2) = ZC[<index>];
[2] restorebind(LAFP);    /* remove bindings */
[3] LAFP = LAFP->ptrval; /* restore local  frame pointer */
[4] [ goto <next>;]
```

fig. c-5

In the above three code sequences, the portion excluding the first statement will be called 'continuation code' and they can be generated at many points in the translator.

5.5.2 VARIABLE REFERENCE

If a variable is declared special, its value-cell has to be accessed. The code is

```
[1] *(LAFP + <dest-offset>) = <var-name>.symval->valcell;
[2] [<continuation code>]
```

fig. c-6

If the variable is unspecial, it is accessed on the stack. The code is

```
[1] *(LAFP + <dest-offset>) = *(LAFP + <var-offset>);
[2] [<continuation code>]
```

fig. c-7

If a variable which has not been declared special, is free at the position where it is referenced, the translator gives a warning and goes ahead treating it as special. In the presen

function every time a reference to it is made, it is explicitly declared as 'extern'. After the translation of the present function is over, the extern declaration is done just after the code for the present function and no more explicit declaration is required. The code to refer a free variable which is not declared as special is

```
{
[1]  extern OBJ <var-name>;
[2]  *(LAFP + <dest-offset>) = <var-name>.symval->valcell;
[3]  [<continuation code>]
}
```

fig. c-8

5.5.3. TRANSLATION OF LISTS (FUNCTION APPLICATIONS)

When the expression is a list, the car of the list has to be a valid functional object which will be applied. So, if the expression passed to tran_exp is a list, it first checks the first element of the list. If it denotes a special function like let, cond etc. which is open-coded, the corresponding open-coder is called. Otherwise, if the first object is a symbol, the declaration for its type is checked. This object can be a list also, which is a valid functional object. Accordingly, 'tran-lambda-call', 'tran-nlambda-call', 'tran-lexpr-call' or 'tran-macro-call' is called if the first object of the list is a lambda, nlambda, lexpr or macro. The default type is taken to be compiled lambda.

Tail recursive calls are handled in a special way. For a tail-recursive call no new frame is created. The arguments are first created on stack and then they are shifted downwards so

that the arguments start just after the current frame pointer. If the same function calls itself, just a jump to the beginning of the function code (BEGIN) is done. 'BEGIN' appeares after the code for saving old values of parameters. So, old values are not saved again and again.

5.5.3.1. LAMBDA CALL

For calling a lambda, the arguments have to be first evaluated and put into proper places of the frame of the callee and then the lambda has to be applied. There can be three diffrent cases of lambda application. The car of the list can be 1) a symbol which is the name of a compiled lambda function, 2) a symbol which is the name of an interpreted lambda or which has a binding to a lambda expression, 3) a list which is a lambda expression.

In the first case, the function is directly called by its name after evaluating the arguments and making the frame. When the call is not tail-recursive, the code looks like

```
[1] <code to evaluate arguments on stack>;
[2] AFP = L1AFP = (LAFP + <offset of new frame>);
                        /* L1AFP is the new frame pointer */
[3] L1AFP->ptrval = LAFP;          /* store old frame pointer */
[4] (L1AFP - 2)->fields.tailf = NOTAIL;   /* put flag to tell
          the callee that the call is not tail-recursive */
[5] (L1AFP - 2)->fields.argcnt = <argno>; /* put arg. count */
[6] F_<function name>();                  /* call function */
[7] [goto <next>;]
```

fig. c-9

If the call is tail-recursive and the same function calls itself, the code is

```
[1] <code to evaluate argume
[2] shift_args(<offset>, <ar
```

```
                                    /* shift the arguments */
[3] goto BEGIN;
```

fig. c-10

If a different function is called in tail-recursive fashion, the code is

```
[1] <code to evaluate arguments on stack>;
[2] shift_args(<offset>, <arg. count>, LAFP);
[3] (LAFP - 2)->fields.tailf = TAIL2;      /* put flag to tell
                 the callee that the call is tail-recursive */
[4] (LAFP - 2)->fields.argcnt = <argno>; /* put arg. count */
[5] AFP = LAFP;                          /* update frame pointer */
[6] F_<function name>();                          /* call function */
[7] return;                  /* last expression of the caller */
      or
    LAFP = AFP;
    [ goto <next>;]                                /* otherwise */
```

fig. c-11

In the second case, the application of the function has to be done by the evaluator. Here a call to the evaluator is generated in mode 2. Instead of "F_<function name>();" shown in figures c-9 and c-11, the corresponding code generated is "eval(2, ZC[<index>]);", where ZC[<index>] is the symbol representing the interpreted function.

In the third case, the lambda object is also translated and the code for it appears just after the code for evaluation of arguments. For a non-tail-recursive call, the code is

```
[1] <code to evaluate arguments on stack>;
[2] (LAFP + <offset of the new frame>)->ptrval = LAFP;
[3] LAFP = LAFP + <offset of the new frame>;
[4] (LAFP - 2)->fields.tailf = NOTAIL;        /* put flag to
              indicate that the call is not tail-recursive */
[5] <code for lambda binding the parameters of lambda>;
[6] <code for the body of the lambda>;
```

fig. c-12

If the call is tail-recursive, the code is as shown below.

The control stays within the same function. So, no shifting of arguments is done and the variables on the stack are still accessible from the new lambda body.

```
[1] <code to evaluate arguments on stack>;
[2] (LAFP - 2)->fields.tailf = TAIL2;      /* put flag to
              indicate that the call is tail-recursive */
[3] <code for lambda binding the parameters of lambda>;
[4] <code for the body of the lambda>;
```

fig. c-13

Some built-in functions do not produce any recursive call. For these functions, the frame manipulation is never tail-recursive.

5.5.3.2. NLAMBDA CALL

In case of an nlambda call, the argument is the unevaluated cdr of the list producing the call. In the above code sequences c-9 to c-13, we substitute "<code to evaluate arguments on stack>" by "*(LAFP + <offset>) = ZC[<index>];" to get the corresponding code for nlambda call. ZC[<index>] is the argument to nlambda.

5.5.3.3. LEXPR CALL

For a lexpr call, the generated code is identical to that of a lambda call. But, here the argument count must be put in the callee's frame, whereas for a lambda call it may be omitted as the functions do not check the number of arguments passed to them.

5.5.3.4. MACRO CALL

While translating a macro call, if the macro definition is

already available, the macro call is expanded and the new expression is translated. There may be more than one macro expansion

If the macro definition is not available, code is generated to expand the call at run time. The code is shown below.

```
[1] <put the macro call expression as arg to macro>;
[2] AFP = L1AFP = (LAFP + <offset of new frame>);
                          /* L1AFP is the new frame pointer */
[3] L1AFP->ptrval = LAFP;          /* store old frame pointer */
[4] (L1AFP - 2)->fields.tailf = NOTAIL;   /* put flag to tell
              the callee that the call is not tail-recursive */
[5] (L1AFP - 2)->fields.argcnt = 1;        /* put arg. count */
[6] F_<function name>();             /* call to compiled macro */
          or
    eval(2, ZC[<index>]);        /* call to interpreted macro */
[7] *(LAFP + <dest-offset>) = eval(1, *(LAFP +<dest-offset>));
          or
    *(LAFP - 2) = eval(1, *(LAFP +<dest-offset>));
                                  /* tail-recursive macro call */
[8] [<continuation code>]
```

fig. c-14

5.5.3.5. OPEN-CODING

A number of special functions are open-coded by the translator. This allows to produce optimized code for those functions after analyzing them properly.

5.5.3.5.1. car

'Car' returns the content of the car field of a cons cell The translator translates a 'car' expression into a simple C expression to extract the car field. Though the error checking done by the in-built function car is sacrificed, one function call is saved by this open-coding. The code produced is

```
[1] <code to evaluate the argument into dest-offset>;
[2] if((LAFP + <dest-offset>)->val != NIL)
      *(LAFP + <dest-offset>) =
                (LAFP + <dest-offset>)->lisval->ca;
    [else rval.val = NIL;]
```

```
                                        /* for tail-recursion */
    [<continuation code>;]
```

fig. c-15

5.5.3.5.2. cdr

'Cdr' is also open-coded in the same way as car. The code for a 'cdr' expression is

```
[1] <code to evaluate the argument into dest-offset>;
[2] if((LAFP + <dest-offset>)->val != NIL)
      *(LAFP + <dest-offset>) =
              (LAFP + <dest-offset>)->lisval->cd;
    [else rval.val = NIL;]        /* for tail-recursion */
    [<continuation code>;]
```

fig. c-16

5.5.3.5.3. cond

The cond expressions are translated into a number of statements which consists of code for evaluation of predicates of clauses, conditional jump statements and code for evaluation of the bodies of clauses. A cond expression is passed to 'tran_cond' for generating the code, which in turn calls 'tran_clause' and 'tran_pred' to translate different parts. If none of the clauses can be evaluated, the code returns nil. For translating a predicate it is provided with two labels: 1) a 'false-label' which is generally the label of the code for the next predicate and 2) a 'true-label' which is the label of the code for the the expressions in the clause following the predicate. If the predicate is the only expression in a clause 'true-label' is the label of the code to be executed after the cond expression. If the predicate returns a non-nil value, the

code at true-label is executed. Otherwise, the code for the nex predicate is executed.

The translator open-codes 'and', 'or' and 'not' expressions. The translation of those are dispatched to corresponding open-coder functions 'tran_and', 'tran_or' and 'tran_not' by 'tran_pred'. These open-coders are also provided with true_label and false_label to produce optimized conditional branch statements. Two more paramters, 'true_jump' and 'flase_jump' are also passed while 'and', 'or' and 'not' expressions are translated. These are flag which indicate whether a branching is necessary if the code evaluates to a non-nil value or nil. The code produced for a cond expression is

```
        <code to evaluate 1st predicate>;
        if((LAFP + <dest-offset>)->val == NIL)
              goto <label for 2nd pred>;  /* false-label */
        [<continuation code>;] /* if clause body is absent */
<label for 1st clause body>:                /* true-label */
        <code to evaluate clause body>;
        <continuation code>;
<label for 2nd predicate>:
        .
        .
        .
        .
<label for nth predicate>:
        <code to evaluate nth predicate>;
        if((LAFP + <dest-offset>)->val == NIL)
              goto <label to return nil>; /* false-label */
        [<continuation code>;]
<label for nth clause body>:                 /* true-label */
        <code to evaluate clause body>;
        <continuation code>;
<label to return nil>:
        *(LAFP + <dest-offset>) = NIL;
        [<cotinuation code>;]
```

fig. c-17

5.5.3.5.4. and

```
        <code to evaluate 1st exp into dest-offset>;
        [if((LAFP + <dest-offset>)->val != NIL)
              goto <true-label>;]
<label for 2nd exp>:          /* false-label for 1st exp */
        <code to evaluate 2nd exp into dest-offset>;
        [if((LAFP + <dest-offset>)->val != NIL)
              goto <true-label>;]
        .
        .
        .
        .
<label for nth exp>:          /* false-label for (n-1)th exp */
        <code to evaluate nth exp into dest-offset>;
        [if((LAFP + <dest-offset>)->val != NIL)
              goto <true-label>;]
        [<continuation code>;]
```

fig. c-19

5.5.3.5.6. not

Translation of a 'not' expression produces code to compute the argument of 'not' and a statement to do a logical not operation on the result. The code for a 'not' expression is

```
<code to evaluate exp into dest-offset>;
(LAFP + <dest-offset>)->val =
                  !(LAFP + <dest-offset>)->val;
[<continuation code>;]
```

fig. c-2Ø

5.5.3.5.7. let

Translation of a let expression produces code for computation of the initialization expressions followed by the code to bind variables to the computed values. If the let is not the last expression of the enclosing block (prog or lambda body or another let), then statements to produce a new frame is created. The code for the latter case is

```
{
    <code to compute 1st initialization>;
    .
```

```
    .
    .
    <code to compute nth initialization>;
    (LAFP + <new frame offset>)->ptrval = LAFP;
                          /* create new frame */
    LAFP = LAFP + <new frame offset>;
    (LAFP - 2)->fields.tailf = NOTAIL;
                         /* not tail-recursive */
    (LAFP - 1)->bsptr = BSP;     /* store BSP */
    [<code to bind 1st variable>;]
            /* bind 1st variable if special */
    .
    .
    .
    [<code to bind nth variable>;]
            /* bind nth variable if special */
    <code for the body of the let>;
}
```

fig. c-21

If the let is tail-recursive, the generated code is

```
{
    <code to compute 1st initialization>;
    .
    .
    .
    <code to compute nth initialization>;
    (LAFP - 2)->fields.tailf = TAIL2;
                            /* tail-recursive */
    [<code to bind 1st variable>;]
            /* bind 1st variable if special */
    .
    .
    .
    [<code to bind nth variable>;]
            /* bind nth variable if special */
    <code for the body of the let>;
}
```

fig. c-22

5.5.3.5.8. prog

In prog all the parameters are first initialized to nil. Then, the prog body is executed. Any symbol in the prog body is converted into a label. Inside a prog body 'return' and 'go' can be done. If a prog expression is not tail-recursive, code to make

a new frame is created. The code produced is

```
{
    (LAFP + <new frame offset>)->ptrval = LAFP;
                          /* create new frame */
    LAFP = LAFP + <new frame offset>;
    (LAFP - 2)->fields.tailf = NOTAIL;
                         /* not tail-recursive */
    (LAFP - 1)->bsptr = BSP;     /* store BSP */
    [<code to bind 1st variable to nil>;]
            /* bind 1st variable if special */
    .
    .
    .
    [<code to bind nth variable to nil>;]
           /* bind nth variable if special */
    <code for the body of the prog>;
}
```

fig. c-23

If a prog expression is tail-recursive, the corresponding code is

```
{
    (LAFP - 2)->fields.tailf = TAIL2;
                          /* tail-recursive */
    [<code to bind 1st variable to nil>;]
            /* bind 1st variable if special */
    .
    .
    .
    [<code to bind nth variable to nil>;]
           /* bind nth variable if special */
    <code for the body of the prog>;
}
```

fig. c-24

5.5.3.5.9. go

The argument to a 'go' has to be a symbol though for an interpreted go expression, it can be anything which evaluates to a symbol. Before a jump to a label can be done, all the frames for surrounding lets and progs (inside the body of the prog which has got the target symbol), have to be removed. The target of a

go can be in an outer prog body also. The code for a go expression is

```
LAFP = LAFP->ptrval->...->ptrval; /* remove all but
                                     last surrounding frame */
restorebind(LAFP); /* restore the bindings using the
     frame pointer to the last frame to be removed */
LAFP = LAFP->ptrval;       /* remove the last frame */
goto L_<symbol in the go expression>;
```

fig. c-25

5.5.3.5.1Ø. return

'Return' can be done from within a prog body. The evaluated argument of a return expression is returned as the value of innermost enclosing prog. Like in case of go, frames for the surrounding let expressions are to removed. The code for a return expression is

```
[1] <code to compute the argument of return>;
[2] rval = *(LAFP + <dest-offset>); /* store result in rval */
[3] LAFP = LAFP->ptrval->...->ptrval;      /* remove all frames
                                     produced in the prog body */
[4] restorebind(LAFP);         /* restore the bindings using the
                                     pointer to the prog frame  */
[5] *(LAFP - 2) = rval; /* put return value at proper place */
[6] <continuation code>;            /* remove the prog frame */
```

fig. c-26

5.5.3.5.11. setq

Setq changes the value of a variable in the current environment. If the variable is special, the value is put in the value-cell of the variable and it can be accessed from any function. The code is

```
[1] <code to compute 2nd argument of setq into dest-offset>;
[2] <variable name>.symval->offset = *(LAFP + <dest-offset>);
[3] [<continuation code>;]
```

fig. c-27

If the variable is unspecial, the value is put on the stack at the position assigned to the variable. It can not be accessed from other functions. The code is

```
[1] <code to compute 2nd argument of setq into dest-offset>;
[2] *(LAFP + <var-offset>) = *(LAFP + <dest-offset>);
[3] [<continuation code>;]
```

fig. c-28

5.5.3.5.12. arg

'Arg' in a lexpr body returns an argument passed to the lexpr. The pointer to the lexpr frame is available in the variable 'LEXFP' local to the code for the lexpr. If there is no argument to 'arg', the number of arguments passed to the lexpr is returned. This number is available in a variable 'vn' which is initialized after entering a lexpr body. The code for an 'arg' expression, when no argument is present, is

```
[1] *(LAFP + <dest-offset>) = vn;  /* get number of args */
[2] <continuation code>;
```

fig. c-29

If there is any argument,

```
[1] <code to compute argument to 'arg' into dest-offset>;
[2] *(LAFP + <dest-offset>) =
          *(LEXFP + *((LAFP + <dest-offset>)->fixval));
[3] [<continuation code>;]
```

fig. c-30

5.5.3.5.13. setarg

'Setarg' in a lexpr body changes the content of a location on the stack, where an argument passed to the lexpr is residing. The code for a setarg expression is shown below.

```
[1] <code to compute 2nd arg into dest-offset>;
```

```
                        /* compute the value to be put */
[2] <code to compute 1st arg into dest-offset + 1>;
        /* compute offset of the argument to be set */
[3] *(LEXFP + *((LAFP + <dest-offset + 1>)->fixval)) =
                                  *(LAFP + <dest-offset>);
                                    /* set the argument */
[4] [<continuation code>;]
```

fig. c-31

5.5.3.5.14. Integer functions

Some arithmetic functions +, -, *, /, =& etc. take only fixnum arguments. As the type of the evaluated arguments are known beforehand, these expressions can be translated efficiently by using the C built-in operators in the translated code. The code for a '+' expression is shown below.

```
<code to evaluate 1st argument into dest-offset>;
   .
   .
   .
<code to evaluate nth argument into (dest-offset + n - 1)>;
*(LAFP + <dest-offset>) =
   makefixnum(*((LAFP + <dest-offset>)->fixval) +
              *((LAFP + <dest-offset> + 1)->fixval) +
              .............    +
              *((LAFP + <dest-offset> + n - 1)->fixval);
[<continuation code>;]
```

fig. c-32

Other functions like '-', '*', '/' are also treated in the same way.

Integer predicates like '=&', '>&', '<&' are also translated to produce efficient code. The code for '=&' is

```
<code to evaluate 1st argument into dest-offset>;
<code to evaluate 2nd argument into (dest-offset + 1)>;
(LAFP + <dest-offset>)->val =
  *((LAFP + <dest-offset>)->fixval) ==
        *((LAFP + <dest-offset> + 1)->fixval);
[<continuation code>;]
```

fig. c-33

Let us consider the translation of two functions fact1 and fact2 to compute the factorial of a number. While fact1 is not tail-recursive, fact2 calls aux_fact and aux_fact is tail-recursive. All the variables are unspecial.

```
 ; following are the Lisp functions.
 (def fact1(lambda(x)
             (cond[(=& x Ø) 1]
                  [t (* x (fact1 (1- x)))])))
 (def fact2(lambda(x)
             (aux_fact x 1)))
 (def aux-fact(lambda(x prod)
                (cond[(=& x Ø) prod]
                     [t (aux-fact (1- x) (* x prod))])))
/* following are the translated functions in C */
extern OBJ fact1;
F_fact1(){
        register OBJ *LAFP = AFP, *L1AFP ;
        OBJ  rval;
        if((LAFP - 2)->fields.tailf == NOTAIL)
                (LAFP - 1)->bsptr = BSP;
BEGIN:
        *(LAFP + 2) = *(LAFP + 1); /* copy x */
L4:
        (LAFP + 3)->fixval = SMALL_FIX_AR + Ø; /* copy Ø */
L5:
        /* check equality of x and Ø */
        (LAFP + 2)->val = *((LAFP + 2)->fixval) ==
                          *((LAFP + 3)->fixval);
L3:     /* if not equal go to L1 */
        if((LAFP + 2)->val == NIL) goto L1;
L2:     /* return 1 */
        (LAFP - 2)->fixval = SMALL_FIX_AR + 1;
        restorebind(LAFP);
        CSP = LAFP - 1;
        AFP = LAFP->ptrval;
        return;
L1:
L7:
        *(LAFP + 2) = *(LAFP + 1); /* copy x */
L8:
        *(LAFP + 6) = *(LAFP + 1); /* copy x */
L11:    /* subtract 1 from x */
        *(LAFP + 6) = makefixnum(*((LAFP + 6)->fixval) - 1);
L1Ø:    /* call fact1 to compute fact(x - 1) */
        AFP = L1AFP = (LAFP + 5);
        L1AFP->ptrval = LAFP;
        (L1AFP - 2)->fields.tailf = NOTAIL;
        (L1AFP - 2)->fields.argcnt = 1;
```

```
        F_fact1();
L9:     /* compute x * fact(x - 1); */
        *(LAFP - 2) = makefixnum(*((LAFP + 2)->fixval) *
                                 *((LAFP + 3)->fixval));
        restorebind(LAFP);
        CSP = LAFP - 1;
        AFP = LAFP->ptrval;
        return;
L6:
        ;
}

extern OBJ fact2;
F_fact2(){
        register OBJ *LAFP = AFP, *L1AFP ;
        OBJ  rval;
        if((LAFP - 2)->fields.tailf == NOTAIL)
                (LAFP - 1)->bsptr = BSP;
BEGIN:
        *(LAFP + 2) = *(LAFP + 1); /* copy x */
L1:
        (LAFP + 3)->fixval = SMALL_FIX_AR + 1; /* copy 1 */
L2:     /* tail-recursive call of type 2, shift arguments */
        shift_args(2,2,LAFP);
        (LAFP - 2)->fields.tailf = TAIL2;
        (LAFP - 2)->fields.argcnt = 2;
        AFP = LAFP;
        F_aux_fact(); /* call aux_fact to compute fact(x) */
        return;
}

extern OBJ aux_fact;
F_aux_fact(){
        register OBJ *LAFP = AFP, *L1AFP ;
        OBJ  rval;
        if((LAFP - 2)->fields.tailf == NOTAIL)
                (LAFP - 1)->bsptr = BSP;
BEGIN:
        *(LAFP + 3) = *(LAFP + 1); /* copy x */
L4:
        (LAFP + 4)->fixval = SMALL_FIX_AR + 0; /* copy 0 */
L5:     /* check equality of x and 0 */
        (LAFP + 3)->val = *((LAFP + 3)->fixval) ==
                          *((LAFP + 4)->fixval);
L3:
        if((LAFP + 3)->val == NIL) goto L1;
L2:     /* x is equal to 0, return prod */
        *(LAFP - 2) = *(LAFP + 2);
        restorebind(LAFP);
        CSP = LAFP - 1;
        AFP = LAFP->ptrval;
        return;
L1:
L7:
```

```
        *(LAFP + 3) = *(LAFP + 1); /* copy x */
L9:     /* compute (x - 1) */
        *(LAFP + 3) = makefixnum(*((LAFP + 3)->fixval) - 1);
L8:
        *(LAFP + 4) = *(LAFP + 1); /* copy x */
L11:
        *(LAFP + 5) = *(LAFP + 2); /* copy prod */
L12:    /* compute (x * p) */
        *(LAFP + 4) = makefixnum(*((LAFP + 4)->fixval) *
                                 *((LAFP + 5)->fixval));
L1Ø:    /* tail-recursive call of type 1, shift arguments */
        shift_args(3,2,LAFP);
        goto BEGIN; /* jump to BEGIN without making further
                       recursive call to aux_fact        */
L6:
        ;
}

fact_init_fns()
{
        fact1.symval->disc = CLAMBDA;
        fact1.symval->func.fval = F_fact1;
        fact2.symval->disc = CLAMBDA;
        fact2.symval->func.fval = F_fact2;
        aux_fact.symval->disc = CLAMBDA;
        aux_fact.symval->func.fval = F_aux_fact;
}
```

5.6. LISPC: THE TRANSLATOR AS A PACKAGE

The translator has been put into a package named "lispc", which can be used conveniently. The command to run the translator is

lispc [options] [files]

The translator recognises a number of options. The options are typed anywhere on the command line. Lisp source files should end with '.l'. Three types of files can be given on the command line, ending with '.l', '.c' and '.o'. While the first type of files are source files, files ending with '.c' are C files translated already and files ending with '.o' are compiled files produced by the C compiler 'cc'.

The translator works in three stages. In the first phase, it translates the Lisp source files and produces C files. In the second phase, it does a source level linking of the C files. In the third phase, it calls the C compiler to compile the C files.

Lispc recognises two options. They are as follows:

Y The translator stops after the first stage i.e. after translating the source files.

Z The translator stops after the second stage i.e. after the source level linking.

If none of the above options are present and no errors occurred during earlier stages, the C compiler is called. Any options given in the command line, except the two mentioned above are passed to the C compiler. So, users can give proper options to be passed to 'cc'.

5.6.1. TRANSLATION OF FILES

All the files ending with '.l' are taken up from the command line and translated one by one. At the end of each generated C file two functions are put: 'init_cons_<file name>' and 'init_fns_<file name>'. These functions are called by the C function 'main' to initialize the Lisp system. The first function is called to initialize the constants present in a Lisp file. The second function is called to put the function types and pointers to the compiled functions for all the functions present in a file in corresponding symbol entries. This information is necessary to allow the evaluator to call the compiled functions.

All the special variables are global and can be accessed by any function in any file. So, they must be declared globally. As a file is translated, all the special variable names and function names are put in a file ending with '.hd'. If a file x.l is translated, the C code goes to x.c and the global declarations go to x.hd. These '.hd' files are used in the next phase.

5.6.2. SOURCE LEVEL LINKING

If no error has occurred in the first phase, the translator checks all the '.hd' files produced in the first phase and the '.hd' files corresponding to the '.c' and '.o' files present in the command line. It produces a C file named 'lispmain.c' which consists of all the global declarations followed by a C function 'main'. In 'main', statements are put to initialize the Lisp system and to initialize the compiled code by 'init_cons_<file name>' and 'init_fns_<file name>' for all the files.

After the initialization statements, a call to the function 'lmain' is generated. This is the name of the main function in Lisp (the top level function) present in some source file. So, 'main' executes 'lmain' and exits. If 'lmain' is not present in any source file, it is taken from the Lisp library by the C complier. This 'lmain' consists of an interpreter read-eval-print loop. So, the 'main' starts behaving as an interpreter.

5.6.3. CALLING C COMPILER

If no error occurs in the second phase and "Y" or "Z" options are not present, the C compiler 'cc' is called. The command line for cc consists of all the options other than "Y" and

"Z" present in 'lispc' command line, all '.c' files produced in the first phase, and all '.c' and '.o' files present in the 'lispc' command line. The C compiler produces the final object code which can be run.

CHAPTER 6

THE LISP READER

The Lisp function 'read' takes the input from a stream of characters and converts it into a Lisp expression. The read operation is table driven and the table used is called the 'readtable'. The 'print' function converts a Lisp expression into a stream of characters. The readtable consists of a number of syntax classes and each character is assigned exactly one syntax class.

6.1. SYNTAX CLASSES

The readtable describes how the reader and the printer should treat each of the 128 ASCII characters. Each character belongs to a syntax class and the properties of the syntax class determines the actions to be taken. Each syntax class has three properties:

character class - It tells what the reader should do when the syntax class has this property. There are twenty one character classes and this number is fixed. They will be discussed later.

separator - Four types of tokens can have an arbitrary length: number, symbol without delimiters, symbol with delimiters (escaped), and string. The reader can easily determine when it has come to an end of one of the last two types - by looking at the symbol delimiter or at the string delimiter respectively. When the reader is reading any one of the first two types, it

ter is encountered whose character class is cseparator or whose syntax class has the separator property. Next, the reader goes into the processing phase.

If the character class of a character which stopped the scanning is not ccharacter, cnumber, cperiod or csign, the reader processes the character immediately. The character classes csingle-macro, csingle-splicing-macro and csingle-infix-macro will act like ccharacter if the following character does not have a separator property. In the processing phase, symbols, numbers and strings are allocated space. If the token has a single character and any read-macro is associated with the character, the read-macro is called. The processing which is done for different character classes is described in detail in the next section.

6.3. CHARACTER CLASSES

The properties of different character classes are described below.

ccharacter - A normal character which can be present in any symbol or string.

cnumber - This type is a digit. The syntax for an integer is a string of cnumber characters. The syntax for a floating point number is either zero or more cnumber's followed by a cperiod and then followed by one or more cnumber's. A floating point number may also be an integer or floating point number followed by 'e' or 'E', an optional '+' or '-' and then zero or more cnumber's.

csign - A leading sign for a number.

cleft-paren - A left parenthesis which means the starting of a

list.

cright-paren - A right-parenthesis which means the end of a list.

cleft-bracket - A left bracket which tells the reader that it should start forming a list.

cright-bracket - A right bracket which finishes the formation of the current list and all enclosing lists until the reader finds one which begins with a cleft-bracket or until it reaches the top level list.

cperiod - The period is used to separate elements of a cons cell. It is also used in numbers as described earlier.

cseparator - This separates tokens. During the scanning process these characters are passed over.

csingle-quote - This causes the reader to be called recursively and the list (quote <value read>) is returned.

csymbol-delimiter - This causes the reader to begin collecting characters and to stop only when another csymbol-delimiter is seen. The only way to escape a csymbol-delimiter within a symbol name is with a cescape character. The collected characters form the print name of a symbol.

cescape - This causes the next character read to be treated as a character in the vcharacter syntax class which has the character class ccharacter and does not have any separator property.

cstring-delimiter - This causes the reader to act in the same way as for csymbol-delimiter. The result is returned as a string.

csingle-character-symbol - This returns a symbol whose print

name is the single character collected by the reader.

cmacro - The reader calls the macro function associated with this character passing it no arguments. The result of the macro is added to the structure the reader is building.

csplicing-macro - A csplicing-macro differs from a cmacro in the way the result is incorporated in the structure the reader is building. This macro must return a list and the reader acts as if it read each element of the list without the surrounding parenthesis.

csingle-macro - If the next character is a cseparator or a character with the separator property, then the macro associated with the character is called and it behaves like a cmacro. Otherwise, it acts like a ccharacter.

csingle-splicing-macro - This is invoked like a csingle-macro. However, the result is spliced like a csplicing-macro.

cinfin-macro - This differs from a cmacro in that the macro function is passed a form representing what the reader has already read so far for building the current list. The result of the macro replaces what the reader has read so far.

csingle-infix-macro - This behaves like a cinfix-macro. But it is invoked like a csingle-macro.

cillegal This causes the reader to signal an error if read.

6.4. STANDARD READTABLE

The standard readtable consists of twenty one syntax classes. The properties of those syntax classes and the characters in those are shown below. The three types of escape properties - escape-always, escape-when-first and escape-when-unique

have been shown as 1, 2 and 3 respectively.

syntax class	character-class	separator	escape
vcharacter	ccharacter	no	no
vnumber	cnumber	no	no
vsign	csign	no	no
vleft-paren	cleft-paren	yes	1
vright-paren	cright-paren	yes	1
vleft-bracket	cleft-bracket	yes	1
vright-bracket	cright-bracket	yes	1
vperiod	cperiod	no	3
vseparator	cseparator	yes	1
vsingle-quote	csingle-quote	yes	1
vsymbol-delimiter	csymbol-delimiter	no	1
vescape	cescape	no	1
vstring-delimiter	cstring-delimiter	no	1
vsingle-character-symbol	csingle-character-symbol	yes	no
vmacro	cmacro	yes	1
vsplicing-macro	csplicing-macro	yes	1
vsingle-macro	csingle-macro	no	3
vsingle-splicing-macro	csingle-splicing-macro	no	3
vinfix-macro	cinfix-macro	yes	1
vsingle-infix-macro	csingle-infix-macro	no	3
villegal	cillegal	yes	1

Properties of syntax classes in standard readtable

syntax class	characters in the syntax class
vcharacter	A-Z a-z ^H ! # $ % & * / : < = > ? @ ^ _ { } ~
vnumber	Ø-9
vsign	+ -
vleft-paren	(
vright-paren	)
vleft-bracket	[
vright-bracket	]
vperiod	.
vseparator	^I-^M esc space
vsingle-quote	'
vsymbol-delimiter	|
vescape	\
vstring-delimiter	"
vsingle-character-symbol	none
vmacro	` ,
vsplicing-macro	;
vsingle-macro	none
vsingle-splicing-macro	none
vinfix-macro	none
vsingle-infix-macro	none
villegal	^@-^G ^N-^Z ^\-^_ rubout

Characters in the syntax classes of the standard readtable

6.5. CHARACTER MACROS

Character macros are executed during the reading process. The value returned by a character macro may or may not be used by the reader, depending upon the type of the macro and the value returned. Character macros can be attached to a character by the 'setsyntax' function.

6.5.1. TYPES OF CHARACTER MACROS

There are three types of character macros - normal, splicing and infix.

6.5.1.1. NORMAL

A normal macro is passed no arguments. The value returned by a normal macro is simply used by the reader as if it read the value itself.

6.5.1.2. SPLICING

The value returned from a splicing macro must be a list or nil. If the value is nil, the value is ignored and the invokation does not register any effect on the reading process. Otherwise, the reader acts as if it read each object in the list. If the reader is reading at the top level, the returned list should not have more than one element.

6.5.1.3. INFIX

If the reader invokes an infix macro while constructing a list, it passes a conc structure representing what has been read so far. The value returned should be a conc structure and the

car of it replaces the list of elements already read by the reader. If the macro is called at top level, it is passed the value nil. If the value returned is nil, it is ignored and the reader continues to read. Otherwise, a conc structure of one element should be returned and the single element is returned as the value read.

6.5.2. INVOCATION OF CHARACTER MACROS

There are three different circumstances in which a character macro is invoked.

1) Invoke always - Whenever the macro character is seen, the macro is invoked. This is accomplished by defining a syntax class using character classes cmacro, csplicing-macro or cinfixmacro and by using the separator property.

2) Invoke when first - The macro is invoked only when the macro character is the first character read after the scanning process. A syntax class for this type of invokation is defined by using character classes cmacro, csplicing-macro and excluding the separator property.

3) Invoke when unique - The macro is invoked only when the macro character is the only character collected in the token collection phase. A syntax class for this is defined by using csingle-macro, csingle-splicing-macro or csingle-infix-macro and excluding the separator property.

6.6. FUNCTIONS TO MANIPULATE THE READTABLE

6.6.1. setsyntax

Setsyntax is used to assign a syntax class to a character. When it is called by '(setsyntax 'symbol 'synclass ['func])', the syntax class 'synclass' is assigned to the first character of 'symbol'. If 'func' is present, this is attached to the same character as a read macro function.

6.6.2. getsyntax

Getsyntax is used to get the syntax class of the first character of the argument which must be a symbol.

6.6.3. add-syntax-class

This can be invoked by '(add-syntax-class 'synclass 'properties)'. It defines a new syntax class 'synclass' with properties present in the list 'properties'. 'properties should contain one character class and may contain one of the three escape properties and the separator property.

CHAPTER 7

INPUT AND OUTPUT FUNCTIONS

A number of functions have been provided for reading from and writing to external devices. All I/O operations are done through the Lisp data type 'PORT'. A port may be opened for reading or writing or both. There are only twenty ports and they are reclaimed when they are closed.

If a port argument is not supplied to a function, the standard input, output and error ports are used. Normally, these three represent the keyboard and the terminal display. Files which are not in the current directory, can be accessed using proper path names. In the following sections different I/O functions have been discussed.

7.1. (infile 'filename)

Infile returns a port for reading the file 'filename'. If the file can not be opened, a error message is given.

7.2. (outfile 'filename ['type])

Outfile returns a port for writing into 'filename'. If 'type' is given and it is a string starting with 'a', the file is opened in the append mode. Otherwise, the opened file is truncated.

7.3. (fileopen 'filename 'mode)

Fileopen opens a file named 'filename' and returns a port

associated with it. 'mode' is a string having one of the following values.

"r" open for reading

"w" create for writing

"a" open for writing at the end of the file (already existing), or create for writing (non-existent).

In addition, each type may be followed by a '+' to have the file opened for reading and writing. "r+" positions the stream at the beginning of the file, "w+" creates or truncates it, and "a+" positions the stream at the end.

7.4. (read ['port ['exp]])

Read returns the next Lisp expression read from 'port' or standard input. On end of file, 'exp' is returned. Internally, the C function 'read_obj' is called to perform the read operation.

7.5. (print 'exp ['port])

'Print' prints the Lisp expression 'exp' on 'port' or standard output. Internally, the C function 'print_obj' is used to perform the print operation.

7.6. (terpr 'port)

Terpr puts the end line character on 'port' or the standard output.

7.7. (load 'filename)

It loads a Lisp source file and the function definitions

present in the file are put in the 'func' fields of the corresponding symbol entries.

7.8. (vi filename), (vil filename)

These functions are used to call the editor 'vi' of the UNIX system to edit 'filename'. The function 'vil' loads the file immediately after editing it.

CHAPTER 8

STORAGE MANAGEMENT

8.1. MEMORY MANAGEMENT SCHEME AND TYPE COMPUTATION

In Lisp,it is the data objects that are typed, not the variables. Variables are typeless and they can have any Lisp object as their value. Often the type of an object has to be determined at runtime. The typing information contributes significantly to the complexity of an implementation.

There is a spectrum of methods for encoding the type of a Lisp object with the following as the two extremes: the typing information can be encoded in the pointer or it can be encoded in the object. If the typing information is encoded in the pointer, then either the pointer is large enough to hold a machine address plus some tag bits or the address itself encodes the type. In the present implementation, the latter one is chosen.

All the data objects are allocated space at run time. In our scheme, the data area is divided into a number of pages of 4K bytes length. Each data type is assigned a number of pages at the time of initialization of the system. All the pages start at 4K bytes boundary and the first byte of each page is used to store the type of the data objects in that page. As the pages are located at 4K byte boundary, if the least significant 12 bits of the address of an object are made zero, we get the address of the page header and that is the place where the type information is

stored. Since the present machine's address space is 32 bits and no bits are left to be used as tag bits, this mechanism is very efficient. It does not use any extra byte to store the type information and it requires minimum computation to get the type information out.

8.2. ALLOCATION OF PAGES

Lisp programs often exhaust the memory very quickly due to the rapid creation of Lisp objects which are needed only temporarily. In the process lot of inaccessible data objects (garbage) are created. These objects must be collected by the storage manager from time to time. To keep track of the allocated space a number of page tables are maintained. For the data types which are not collectable, a single page table for each one of them is kept in the memory.

For the data types which are collectable, two types of pages are created: static and dynamic. Static pages are used to store the constant expressions generated by the compiled code. These are created just after the loading of the system and once allocated, the objects in the static pages are never relocated or reclaimed. On the other hand, the objects in the dynamic pages are relocated and reclaimed. For these two types of pages of each collectable data type two page tables are kept. At present, lists and integers are collectable while others are not.

At the beginning the storage manager takes a big chunk of memory (multiple of 64K bytes or 16 pages) from the system and creates a number of pages by dividing the chunk and aligning the

parts at 4K byte boundaries. A number of pages are allocated for each data type and the corresponding page tables are initialized with the addresses of the page headers. A number of free pages are kept in the free page table to be allocated later on demand. For each page table two indices, one showing the current page from which objects are being allocated and the other showing the total pages in the page table are maintained. As the current page in a page table is exhausted, allocation from the next page is started. If no more pages are available for a non-collectable data type, a new page is allocated from the free page table. To fulfill the demand for new pages, the storage manager may take a new chunk from the system if the demand can not be met by the existing (or non-existing) pages in the free page table.

When the static pages for a collectable data type are exhausted, the same action, as in the case of non-collectable data types is taken. If the dynamic pages for a collectable data type are exhausted, garbage collection for that particular type starts. After collecting all the inaccessible data objects, the storage manager decides whether to allocate more pages or not.

Integers are allocated space in a special way. A whole static page is allocated to store small integers starting from Ø to 1Ø22. If a new integer is created in this range, no new space is created; instead the pointer to the same integer in that static page is returned. It saves lot of space as multiple copies of small integers are not created. Also, the process of storing small integers becomes faster as it needs only the addition of the integer value to the address of Ø.

8.3. GARBAGE COLLECTION

Presently, the garbage collection is done for integers and cons cells. Garbage collection for other data types can also be implemented. It may not be done for strings as it requires considerable overhead.

8.3.1. GARBAGE COLLECTION OF CONS CELLS

PORTLISP uses copying garbage collection strategy. When all the allocated dynamic pages are exhausted, another call to the storage manager to allocate a cons cell initiates the garbage collection process. First all the old pages are removed to a page table called 'garbage page table' and the page table (for dynamic pages) for cons cells is filled up with equal number of new pages from the free page table. After that, the scanning and copying of accessible cons cells start. The copying process is done by calling a function 'cp_list' recursively. After copying a cell, it must be possible to tell when an object is forwarded from the old space. For this, a special flag 'COPIED' is used. When a cons cell is copied, this flag is put in the car field of the old cons cell and the new address to which the cell has been forwarded is put in the cdr part. As a pointer to a cons cell is passed to 'cp_cell', it checks whether the cell is residing in a static page. If it is it returns the old pointer. Otherwise, it accesses the cons cell and checks the car of it. If the flag 'COPIED' is stored there, cp_list returns the cdr of the cell where the new address is available. If the cell is not already copied, it first puts the flag COPIED in the car field, gets a new cell and puts

the address in cdr part. After this if the car and the cdr are lists, cp_list is called recursively to copy them and the returned addresses are put in the car and the cdr of the new cell. Otherwise, the old values of car and cdr are simply copied to the corresponding fields of the new cell. After the copying of car and cdr is over, cp_list returns the new address.

In the process of garbage collection all the cons cells which are accessible are copied. For this all the symbol entries are accessed and their value-cells, function fields and property lists are checked and if they are lists, they are copied. Old bindings are saved on the binding stack and in the fclosure objects. So the binding stack and all the fclosure objects are accessed to find out the accessible cons cells in the dynamic pages. Intermediate results and temporary variables are located on the control stack and the code stack and any lists present on these stacks are also copied to new pages.

After the copying is over, the pages from the garbage page table are moved to the free page table. The storage manager computes the total allocated space and free space for lists. If the free space is below a certain level (computed by amount of free space / amount of total space), new pages are allocated to maintain the level.

Users must be careful about updating any cons cell in a static page destructively. The elements in the static pages should contain pointers to elements in static pages only. These are not forwarded or collected. If any cons cell in the static area has a pointer to an element in the dynamic area, as the

second object is copied to some other location the pointer in the cons cell will point to some undefined or wrong object.

8.3.2. GARBAGE COLLECTION OF FIXNUMS

In case of cons cells we have seen that after a cons cell is copied to a new location, a special flag COPIED and the new address is put in the old cons cell. As an integer is stored using all the 32 bits of a fixnum object there is not enough room left to put a distinguishable flag and a forwarding pointer in the old space allocated for a fixnum. So, fixnums are copied in a special way. When the old dynamic pages are consumed, they are removed to the garbage table and equal number of new pages are put in the page table for fixnum. A fixnum in the ith page in the garbage table is always copied to the first available object in the ith new page. The displacement of the relocated object in the new page is put in place of the old object. This displacement is in the range of Ø - 1Ø22 and recall that no integer in this range can be in a dynamic page. So this does not give rise to any conflict. When an already copied fixnum in the ith old page is visited the displacement value is recognised. The new address is computed using this value and the address of the new ith page, and this address is returned.

CHAPTER 9

CONCLUSION

The major achievement of this project is eliminating the need for a Lisp compiler. We have seen how Lisp expressions are translated into C functions. As C gets compiled efficiently, the final object code produced is quite efficient.

The power of the translator lies in its ability to open-code a number of functions. The open-coder functions analyze the source code properly and produce optimized code. Open-coding often eliminate function calls as in case of car, cdr and integer functions which are used heavily. The interpreter and the translator, both have tail-recursive features. The translator detects all the tail-recursive expressions in a piece of program and produces proper translated C code. This minimises the stack overflow and saves lot of time.

Still there are a number of bottle-necks in the system. Lisp and C are totally different from each other. While Lisp uses dynamic scoping, C uses static scoping. In Lisp whenever a Lisp object is passed to another function or copied from one variable to another, actually the address is passed or copied. In C, the values of variables are passed or copied. The reading process in Lisp is table driven, but in C it is not. For running the translated code and interfacing it properly to the interpreter, different stacks are used. While the control stack is used to make frames for calling functions, passing variables and

returning values, the C stack is used to implement the call and return mechanisms. It is not really possible to push and pop return addresses in C while calling a function. For a tail-recursion of type 2, though the control stack does not grow, the C stack will grow and may lead to stack overflow. For a series of tail-recursive calls of type 2, only one return address is needed to be pushed while the first function call is initiated.

In the translator, it is not possible to handle register allocation for variables and other optimizations possible while generating machine code. We do not have any control on saving of registers during function calling. Some functions like car, cdr can be executed using only one register, and saving of registers is not necessary. If the intermediate language chosen for translation can specify many things about the target machine code, more optimizations can be done. The control stack pointer and the active frame pointers are kept in global variables and they are used for all accesses on the stack. There is no way to keep them in registers permanently.

At present a copying garbage collector is implemented. It is fast, but uses extra storage while doing the garbage collection. It is difficult to implement other types of garbage collectors due to non-availability of extra bits for usage as tags.

The present implementation can be improved in a number of ways. More open-coder functions can be introduced to increase the efficiency of the code. Presently, every object that is passed or copied is a pointer to the actual object. The integers are also treated in the same way. But the size of an integer and

a pointer to it, both are 4 bytes long. A good amount of storage can be saved if the integers are made immediate type sacrificing one bit to show that it is an immediate integer. If integers are used as immediate, boxing and unboxing of integers can be avoided.

The translator produces code to box the result after evaluation of each arithmetic expression. When there is a nested expression involving many arithmetic functions, the boxing of the result may be done after computing the final result. This will save time and intermediate objects will not be heap-allocated.

The implemented translator will only know the type of arguments to integer functions. Declarations for forcing types of expressions may be introduced to supply more information to the translator.

At present the garbage collection is done only for cons cells and integers. The same for other types can be implemented.

Presently all the data types available in FRANZ LISP are not implemented. The system can be further enhanced by the implementation of 'bignum', 'array', 'hunk' and 'vector' data types.

REFERENCES

[Abelson;1985]

Harold Abelson and Gerald Jay Sussman, **Structure and Interpretations of Computer Programs**, MIT Press, 1985.

[Aho;1986]

Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman, **Compilers: Principles, Techniques and Tools**, Addison-Wesley, 1986.

[Allen;1978]

John Allen, **Anatomy of Lisp**, McGraw-Hill, 1978.

[Baker;1978a]

Henry J. Baker, **List Processing in Real Time on a Serial Computer**, Communications of the ACM, 21(4), pp. 280-293, April, 1978.

[Baker;1978b]

Henry J. Baker, **Shallow Binding in Lisp**, Communications of the ACM, 21(7), pp. 565-569, July, 1978.

[Bobrow;1973]

Daniel G. Bobrow and Ben Wegbreit, **A Model and Stack Implementation of Multiple Environments**, Communications of the ACM, 16(10), pp. 591-603, October, 1973.

[Brooks;1982]

Rodney Brooks, Richard P. Gabriel and Guy L. Steele,

Common Lisp Implementation, ACM Symposium on Lisp and Functional Programming, pp. 108-113, 1982.

[Foderaro;1983]

John K. Foderaro, Keith L. Sklower and Kevin Layer, **Franz Lisp Manual,** June, 1983.

[Gabriel;1986]

Richard P. Gabriel, **Performance and Evaluation of Lisp Systems,** MIT Press, 1986.

[Griss;1982]

Martin L. Griss, Eric Benson and Gerald Q. Maguire Jr., **PSL: A Portable Lisp System,** ACM Symposium on Lisp and Functional Programming, pp. 88-97, 1982.

[Hickey;1984]

Tim Hickey and Jacques Cohen, **Performance Analysis of On-the-Fly Garbage Collector,** Communications of the ACM, 27(11), November, 1984.

[Meehan;1979]

James R. Meehan, **The New UCI Lisp Manual,** Lawrence Erlbaum Associates, Publishers, 1979.

[Padget;1985]

Julian Padget and John Fitch, **Closurize and Concentrate,** Twelfth Annual ACM Symposium on Principles of Programming Languages, pp. 255-265, January, 1985.

[Rees;1982]

Jonathan A. Rees and Norman I. Adams, **T: A Dialect of Lisp or, LAMBDA: The Ultimate Software Tool,** ACM Symposium on

Lisp and Functional Programming, pp. 114-122, 1982.

[Steele;1976]

Guy Lewis Steele, Lambda, the Ultimate Declarative, AI Memo 379, MIT Artificial Intelligence Lab., November, 1976.

[Steele;1982]

Guy Lewis Steele, An Overview of Common Lisp, ACM Symposium on Lisp and Functional Programming, pp. 98-107, 1982.

[Steele;1984]

Guy Lewis Steele, Common Lisp the Language, Digital Press, 1984.

APPENDIX

LIST OF IMPLEMENTED FUNCTIONS

DATA STRUCTURE ACCESS FUNCTIONS

car, cdr, cons, list, append, listp, length, nth, nthcdr, nthelem, atom, rplaca, rplacd, nconc, eq, equal, not, null, member, memq, implode, gensym, boundp, symeval, plist, set, setq, explode, assoc, get, putprop, defprop, tconc, lconc, quote.

SPECIAL FUNCTIONS

and, apply, arg, *catch, cond, declare, def, do, eval, fclosure, funcall, go, let, let*, or, prog, return, setarg, *throw.

ARITHMETIC FUNCTIONS

add, +, add1, 1+, diff, -, sub1, 1-, times, *, quotient, /, numberp, fixp, floatp, zerop, onep, plusp, minusp, greaterp, >&, lessp, <&, =, =&, mod, abs.

READER, I/O, EDITOR FUNCTIONS

add-syntax-class, setsyntax, getsyntax, fileopen, infile, outfile, close, read, print, terpr, load, vi, vil.